UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA CIVIL

UM AMBIENTE COMPUTACIONAL ORIENTADO POR OBJETOS PARA ANÁLISE DE ESTRUTURAS APORTICADAS TRIDIMENSIONAIS

Eng. Armando Diório Filho

Orientador: Prof. Dr. José Luiz Antunes de Oliveira e Sousa

Declaro que este exemplar corresponde a versão definitiva da dissertação de mestrado do Engº Armando Diório Filho

Campinas, 23 de julho de 1996

orientador

CAMPINAS - SP - BRASIL

1996

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA CIVIL

UM AMBIENTE COMPUTACIONAL ORIENTADO POR OBJETOS PARA ANÁLISE DE ESTRUTURAS APORTICADAS TRIDIMENSIONAIS

Eng. Armando Diório Filho

Orientador: Prof. Dr. José Luiz Antunes de Oliveira e Sousa

Dissertação apresentada junto à Faculdade de Engenharia Civil da Universidade de Campinas, como parte dos requisitos para obtenção do título de Mestre em Engenharia Civil.

Área de concentração em Estruturas.

CAMPINAS - SP - BRASIL

1996

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Diório Filho, Armando

D623a Um ambiente computacional orientado por objetos para análise de estruturas aporticadas tridimensionais / Armando Diório Filho.--Campinas, SP: [s.n.], 1996.

Orientador: José Luiz Antunes de Oliveira e Sousa.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Civil.

1. Programação orientada a objetos. 2. Análise estrutural (Engenharia). 3. Pórticos estruturais. 4. Método dos elementos finitos. I. Sousa, José Luiz Antunes de Oliveira e. II. Universidade Estadual de Campinas. Faculdade de Engenharia Civil. III. Título.

FOLHA DE APROVAÇÃO

Prof. Dr. José Luiz Antunes de Oliveira e Sousa
Orientador

Prof. Dr. Philippe Remy Bernard Devloo
FEC - UNICAMP

Prof. Dr. Túlio Nogueira Bittencourt
EPUSP - SÃO PAULO

À Sueli e aos meus filhos
Armando, Aline e Alexandre

AGRADECIMENTOS

Ao meu orientador e amigo Prof. Dr. José Luiz Antunes de Oliveira e Sousa, pela oportunidade de pesquisa oferecida e pelo apoio, incentivo e segurança transmitidos.

Ao professor Dr. Philippe Remy B. Devloo pelas dúvidas esclarecidas relacionadas com a linguagem C++.

Aos professores Drs. Francisco Antonio Menezes e José Luiz Fernandes de A. Serra, pelas informações sobre análise de estruturas.

Aos membros da Comissão Examinadora, pela atenção.

Aos colegas de pós-graduação da FEC pela amizada e convívio.

À Faculdade de Engenharia Civil, ao Departamento de Construção Civil, pelas instalações e recursos colocados à disposição.

À Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP, pelo financiamento, na forma de bolsa de estudo, proporcionado para a realização deste trabalho.

Enfim, a todos aqueles que, de qualquer forma, participaram da elaboração deste trabalho.

SUMÁRIO

LISTA DE TABELAS E FIGURAS

LISTAGENS

RESUMO

1. - Introdução 1

2. - A Programação Orientada por Objetos - OOP 7

2.1 - Como Interpretar a OOP 8

2.2 - Elementos da OOP 10

2.2.1 - Classes 10

2.2.2 - Objetos 12

2.2.3 - Métodos 12

2.3 - Propriedades da OOP 13

2.3.1 - Encapsulamento 14

2.3.2 - Herança 14

3. - Elementos Finitos - Formulação 17

3.1 - Apresentação 17

3.2 - Modelo de Formulação Adotado 24

3.3 - Formulação 25

4 - Formulação do Problema Variacional Para um Trecho Genérico de Viga no Espaço Tridimensional e Obtenção da Matriz de Rigidez 30

4.1 - Funcional da Energia Potencial Total 30

4.1.1 - Parcela Devida ao Esforço e Deslocamento Normal 32

4.1.1.a - Função Aproximadora 33

4.1.1.b - Expressão da Energia de Deformação 34

4.1.1.c - Energia Potencial do Trabalho das Forças Externas 36

4.1.2 - Parcela Devido ao Esforço Fletor e Deslocamentos Correspondentes 37

4.1.2.a - Função Aproximadora 38

4.1.2.b - Expressão da Energia de Deformação 43

4.1.2.c - Energia Potencial do Trabalho das Forças Externas 47

4.1.2.c-1) Trabalho realizado pelas forças concentradas P_{fi} 47

4.1.2.c-2) Trabalho Realizado Pela Carga Uniformemente Distribuida Q 48

4.1.2.c-3) Trabalho Realizado Pela Carga Distribuida Linearmente q(x) 49

4.1.3 - Parcela Devido ao Esforço Torçor e Deslocamentos Correspondentes 53

4.1.3.a - Função Aproximadora 54

4.1.3.b - Expressão da Energia de Deformação 55

4.1.3.c - Energia Potencial do Trabalho das Forças Externas 59

4.1.3.c-1) Trabalho Realizado Pelos Momentos de Torção 59

4.2 - Determinação Da Matriz De Rigidez Para Um Elemento De Barra No Espaço Tridimensional 60

5. - As Classes Básicas Do Sistema 65

5.1 - Uma Função Para Erros 66

5.2 - O Problema Armazenado Em Listas 68

5.3 - As Classes Para Vetores 72

5.4 - A Classe TSqMatrix 75

5.5 - As Classes TNode e TNode3d 77

5.6 - As Restrições Aos Deslocamentos 81

5.7 - Os Recalques 84

5.8 - Os Elementos 87

5.8.1 - O Elemento Genérico 87

5.8.2 - O Elemento De Barra 90

5.9 - Os Arquivos De Perfis E As Seções Transversais 95

5.10 - Os Materiais Dos Elementos 106

5.11 - Carregamento : Uma Coletânea De Classes 108

6 - As classes Que Gerenciam O Sistema 120

6.1 - O Modelo (Estrutura) 122

6.2 - As Classes Que Gerenciam Nós 124

6.2.1 - Nós Geométricos 124

6.2.2 - Nós Estruturais 127

6.3 - Gerenciando Materiais e Seções 132

6.4 - Gerenciando As Cargas Do Modelo 137

6.5 - Gerenciando O Modelo Composto Por Barras 143

7 - Exemplos 152

7.1 - Exemplo 1 153

7.2 - Exemplo 2 154

7.3 - Exemplo 3 156

7.4 - Exemplo 4 158

7.5 - Exemplo 5 160

8 - Conclusões 167

8.1 - Introdução 167

8.2 - Avaliação da Utilização da POO 168

8.3 - Análise de Resultados de Simulações 170

8.4 - Sugestões Para Trabalhos Futuros 171

APÊNDICE A - ROTEIRO PARA ELABORAÇÃO DE ARQUIVO DE DADOS 175

Referências Bibliográficas 179

Bibliografia Consultada 181

LISTA DE TABELAS E FIGURAS

Figura 2.1 - Diferença entre OOP e Programação Tradicional 9

Figura 3.1 - Convergência (neste caso "por baixo") da solução aproximada à solução exata 23

Figura 4.1 - Sistema de Eixos do Elemento de Barra (Sistema Local de Coordenadas) 31

Figura 4.2 - Trecho de Uma Viga com Deslocamentos u_1 e u_7 nas Extremidades e Cargas N(x), P_1...P_i 32

Figura 4.3 - Esquema de Variação Linear para as Funções η_1 e η_7 34

Figura 4.4 - Trecho de viga no plano **XY** com deslocamentos $\mathbf{v_2}$, θ_3 ,$\mathbf{v_8}$ e θ_9 nas extremidades e sujeita as cargas concentradas $\mathbf{P_{f1}}$...$\mathbf{P_{fi}}$, uniformemente distribuida **Q** e linearmente distribuida **q(x)** 38

Figura 4.5 -Funções de Interpolação η_2 , η_3 , η_8 e η_9 40

Figura 4.6 - Deformações de um trecho *dx* de um elemento de viga sobre flexão 43

Figura 4.7 - Trecho de Uma Viga com Deslocamentos no plano XZ w_4 , θ_5 , w_{10} e θ_{11} das Extremidades 51

Figura 4.8 - Trecho de Uma Viga com Deslocamentos φ_6 e φ_{12} nas Extremidades no Plano YZ 53

Figura 4.9 - Distribuição de Tenções τ ao longo da Espessura e Orientação da Coordenada C_r 56

Figura 5.1 - Esquema de Uma Lista de Vetores 68

Figura 5.2 - Esquema Para As Classes Que Representam Vetores 73

Figura 5.3 - Relacionamento Entre A Classe TNode E A Classe Derivada TNode3d 80

Tabela 5.1 - *Struct* freedom 83

Tabela 5.2 - Posição dos recalques no objeto fvet_delta da classe TVetFloat 84

Figura 5.4 - Hierarquia entre a classe TElement e suas classes derivadas 88

Figura 5.5 - Ângulo de rotação em torno do eixo local X_m 92

Figura 5.6 - Hierarquia Entre as Classes Que Manipulam Arquivos de Seções 96

Figura 5.7 - Esquema para atribuições e alterações das seções às barras 106

Tabela 5.2 - Posição das forças na variável fvetload 109

Figura 5.8 - A hierarquia entre as classes de tipos de carregamentos 111

Figura 5.9 - Ângulo de inclinação das cargas na análise tridimensional 115

Figura 6.1 - Esquema de Interligação dos Modelos 121

Figura 7.1 - Uma das Estruturas Aporticadas Analisadas na Fase de Eliminação de Erros 153

Tabela 7.1 - Exemplo 1 - Resultados Obtidos Utilizando O Programa Desenvolvido 153

Tabela 7.2 - Exemplo 1 - Resultados Fornecidos Pelo Programa SAP-90 154

Figura 7.2 - Estrutura Aporticada Proposta Para Análise 154

Tabela 7.3 - Exemplo 2 - Resultados Obtidos Utilizando O Programa Desenvolvido 155

Tabela 7.4 - Exemplo 2 - Resultados Fornecidos Pelo Programa SAP-90 155

Figura 7.3 - Exemplo Máximo Cuja Análise Foi Conseguida Com A Utilização De Matriz Cheia 156

Tabela 7.5 - Exemplo 3 - Resultados Obtidos Utilizando O Programa Desenvolvido 157

Tabela 7.6 - Exemplo 3 - Resultados Fornecidos Pelo Programa SAP-90 157

Figura 7.4 - Exemplo De Análise Considerando-se Combinação De Carregamentos 158

Tabela 7.7 - Exemplo 4 - Resultados Obtidos Para a Primeira Combinação Entre Os Carregamentos 159

Tabela 7.8 - Exemplo 4 - Resultados Obtidos Para a Segunda Combinação Entre Os Carregamentos 159

Figura 7.5 - Exemplo de estrutura analisada utilizando matriz *skyline* 163

Tabela 7.9 - Exemplo 5 - Resultados Obtidos Utilizando O Programa Desenvolvido 164

Tabela 7.10 - Exemplo 5 - Resultados Fornecidos Pelo Programa SAP-90 165

LISTAGENS

Listagem 5. 1 - Parte do arquivo error.h 66

Listagem 5. 2 - Parte do arquivo de cabeçalho da classe TList 69

Listagem 5. 3 - Parte do arquivo de cabeçalho da classe TVetor 72

Listagem 5. 4 - Parte do arquivo de cabeçalho da classe TVetInt 73

Listagem 5. 5 - Parte do arquivo de cabeçalho da classe TVetFloat 74

Listagem 5. 6 - Parte do arquivo de cabeçalho da classe TSqMatrix 75

Listagem 5. 7 - Parte do arquivo de cabeçalho da classe TNode 79

Listagem 5. 8 - Parte do arquivo de cabeçalho da classe TNode3d 80

Listagem 5. 9 - Parte do arquivo de cabeçalho da classe TRestricao 82

Listagem 5. 10 - Parte do arquivo de cabeçalho da classe TRecalque 86

Listagem 5. 11 - Parte do arquivo de cabeçalho da classe TElement 89

Listagem 5. 12 - Parte do arquivo de cabeçalho da classe TBarra 91

Listagem 5. 13 - Parte do arquivo de cabeçalho da classe TBarra3d 93

Listagem 5. 14 - Parte do arquivo de cabeçalho da classe TGeneric_Secao 99

Listagem 5. 15 - Método construtor da classe TSecao 100

Listagem 5. 16 - Parte do arquivo de cabeçalho da classe TSecao 102

Listagem 5. 17 - Parte do arquivo de cabeçalho da classe TGrup_Secao 104

Listagem 5. 18 - Parte do arquivo de cabeçalho da classe TMaterial 107

Listagem 5. 19 - Parte do arquivo de cabeçalho da classe TNode_Load 108

Listagem 5. 20 - Parte do arquivo de cabeçalho da classe TGeneric_Load 110

Listagem 5. 21 - Parte do arquivo de cabeçalho da classe TDistrib 112

Listagem 5. 22 - Parte do arquivo de cabeçalho da classe TBar_Load 114

Listagem 5. 23 - Parte do arquivo de cabeçalho da classe TLoad2d 117

Listagem 6. 1 - Parte do arquivo de cabeçalho da classe TModel 122

Listagem 6. 2 - Parte do arquivo de cabeçalho da classe TNode_Manager 125

Listagem 6. 3 - Parte do arquivo de cabeçalho da
classe TModel_Struct_Node_2D 130

Listagem 6. 4 - Esquema de parte do arquivo de cabeçalho da
classe TModel_Struct_Node_3D 132

Listagem 6. 5 - Parte do arquivo de cabeçalho da classe TMat_Manager 134

Listagem 6. 6 - Parte do arquivo de cabeçalho da classe TSec_Manager 136

Listagem 6. 7 - Parte do arquivo de cabeçalho da classe TComb_of_Load 139

Listagem 6. 8 - Parte do arquivo de cabeçalho da classe TLoad_With_Weight 139

Listagem 6. 9 - Parte do arquivo de cabeçalho da classe TLoad2d_Manager 141

Listagem 6. 10 - Sequência de instruções do método Read_Data_File 145

Listagem 6. 11 - Sequência de instruções do método Write_Data_File 145

Listagem 6. 12 - Parte do arquivo de cabeçalho da
classe TModel_With_Bar_2D 147

Listagem 6. 13 - Sequência de instruções do programa principal 150

Listagem 7.1 - Parte do arquivo de cabeçalho da classe TModel_Integrated 162

RESUMO

Esta dissertação descreve o desenvolvimento de um sistema orientado por objetos utilizado em análises de estruturas aporticadas tridimensionais. Este sistema é formado por uma biblioteca de classes, escritas em linguagem de programação C++, as quais podem ser utilizadas em diversas aplicações, permitindo que novas teorias e idéias sejam implementadas com o trabalho concentrado nessas novas potencialidades em desenvolvimento. Isso resulta em uma otimização do tempo e esforços necessários para implementar as novas funcionalidades ao sistema. Um exemplo desta característica da filosofia da orientação por objetos foi observada durante o desenvolvimento deste trabalho no uso de classes para a solução de sistemas de equações lineares desenvolvida, testada e otimizada por outro membro do grupo de pesquisa.

O sistema aqui desenvolvido foi testado e os resultados obtidos foram validados por comparação com um programa comercial de ampla utilização.

Devido às características próprias da filosofia da orientação por objetos e da estrutura do sistema desenvolvido, extenções envolvendo não-linearidade física e geométrica, análise dinâmica, otimização estrutural e outros problemas relacionados a estruturas aporticadas podem ser rapidamente implementados.

ABSTRACT

This dissertation describes the development of an object oriented system for the numerical analysis of tridimensional framed structures. The system consists of a library of object classes, written in C++ language, which are designed to be used in different applications, thus allowing that new theories and ideas be implemented with most of the work concentrated in the new features under development. The result is an optimization of the time and work necessary for the implementation of new funcionalities to the system. An example of this characteristic of the object oriented philosophy observed in the development of this work is the use of classes for the solution of linear systems, developed, tested and optimized by other member of the research group.

The developed system was tested and validated by comparison with a largely used commercial code.

Due to the characteristics inherent to the object oriented philosophy, and the design bases for the developed system, extensions to handle problems involving geometric and physical nonlinearities, dynamic analysis, structural optimization and other problems related to framed structures can be readily implemented.

1 - INTRODUÇÃO

Vários são os usos do computador na engenharia (NAJAFI-[1991]): planejamento, gráficos, administração, análise estrutural entre outros, que permitem aos engenheiros, administradores e pessoas envolvidas nos projetos planejá-los, estimar seus impactos, avaliar informações mais rapidamente e tomar decisões com menor possibilidade de erros.

No desenvolvimento de técnicas computacionais, as dificuldades mais comuns que geralmente são enfrentadas para a obtenção da solução de problemas estruturais referem-se à elaboração de entrada de dados, processamento e visualização de resultados, interação entre usuário e máquina durante o processo de análise, e implementação de novas potencialidades aos programas resultantes. Com isto, um programa que poderia ser muito útil não é suficientemente explorado porque, entre outros fatores, não é receptivo a extensões para o tratamento de outros problemas.

Do ponto de vista de desenvolvimento de pesquisa em grupo envolvendo *software*, é de fundamental importância que o código fonte gerado por um pesquisador seja facilmente entendido e reutilizado por outros em novas aplicações. A programação procedural (ou tradicional), normalmente utilizada no desenvolvimento de aplicações, não tem sido adequada sob este ponto de vista. O código fonte pode ser reutilizado, mas o custo dessa reutilização requer alto investimento de tempo no seu entendimento e readaptação, com prováveis erros em sua manipulação, ou restrições à estrutura de dados impostas pela utilização de bibliotecas de rotinas geradas por terceiros.

Considerando-se a necessidade de introduzir uma nova potencialidade a um código já existente, mesmo que pelo seu idealizador, a tarefa de abrir o código, fazer as alterações que forem necessárias, implementar o novo algoritmo e testar todo o código pode se transformar em um enorme problema, as vezes até maior que a novidade que se deseja introduzir. Assim, em certos casos, acaba-se optando por desenvolver um outro programa que execute a nova potencialidade, perdendo-se um tempo considerável reescrevendo código já escrito e incorrendo-se na possibilidade de que esse novo código seja extremamente específico ou condenando-se o código anterior ao desuso.

Usando-se a linguagem procedural no desenvolvimento de uma pesquisa que inclua a elaboração de um programa de computador, o simples fato das especificações de projeto sofrerem alterações durante o desenvolvimento do mesmo causa enorme transtorno, pois com frequência essas alterações exigem uma readaptação a partir do início do programa e a modificação de funções já testadas, requerendo dispêndio de tempo por parte do programador e incorrendo novamente na possibilidade de introduzir erros em código estável.

Visando superar esse tipo de problema, as linguagens de programação evoluiram chegando hoje ao patamar mais elevado de desenvolvimento (MACKIE[1992]): o paradigma da Programação Orientada por Objetos (**OOP**). Tal filosofia de programação se baseia no fato de as ações (métodos) sobre os dados serem definidas junto com os mesmos, levando ao encapsulamento que caracteriza o objeto (ou mais apropriadamente, sua classe).

Quando um objeto é incluido em uma nova aplicação, apenas os dados necessários para sua criação e utilização e os métodos a que ele responde devem ser de conhecimento público, evitando-se assim a manipulação de sua estrutura de dados por outras porções de códigos distintas daquelas que definem seus métodos. Esse encapsulamento dos dados pelo objeto permite que a manutenção e verificação dos programas sejam facilitadas.

Adicionalmente, a Programação Orientada por Objetos permite que sejam criadas classes derivadas de classes existentes, especializando-as com acréscimo de dados e métodos sem a necessidade de manipular o código já existente, testado e considerado estável. O conceito de classe e herança melhoram o gerenciamento de dados e a modularização (SCHOLZ[1992]).

A aplicação da programação orientada por objeto aos problemas de análise estrutural permite que novas potencialidades sejam adicionadas a um sistema de modo mais simples, com maior rapidez e com menor possibilidade de inclusão de erros em código já testado. Isso é conseguido por intermédio do uso da herança através da qual, normalmente, as classes mais específicas representam as novas potencialidades .

O objetivo deste projeto é o desenvolvimento de uma biblioteca de classes de objetos relacionados a estruturas aporticadas tridimensionais (a análise plana deixa de levar em consideração os efeitos de enrijecimento de certas peças estruturais como lajes, poços de elevadores, etc - LEUNG[1985]). Essa biblioteca deverá constituir a base de um sistema concebido para crescer indefinidamente, permitindo que novas teorias e idéias envolvendo estruturas tridimensionais de edifícios possam ser rapidamente implementadas com a utilização de objetos especializados, exaustivamente testados, levando a uma otimização de tempo e esforço do pesquisador que vier a introduzir novas potencialidades ao sistema. Essas potencialidades a serem introduzidas posteriormente poderão incluir, entre outras, análise geométrica e fisicamente não-linear, análise dinâmica, otimização,

dimensionamento envolvendo diferentes materiais, combinação com diferentes tipos de elementos estruturais como placas e cascas, e, eventualmente, detalhamento da estrutura.

Devido às características deste projeto, a biblioteca de objetos poderá ser utilizada por terceiros que, em princípio, não terão acesso ao código fonte, sendo então de fundamental importância a documentação detalhada de cada classe, com seus métodos e retrições de uso. A falta de uma documentação ou a documentação incompleta de um *software* é um tipo de problema, entre outros, que já foi detectado por EMKIN[1988]; e, com relação aos futuros usuários do ambiente computacional aqui apresentado, vale ressaltar o que McGUIRE[1992] cita em seu trabalho: "o conhecimento do método dos elementos finitos é essencial para a compreensão de uma análise computacional, e esta requer do analista um conhecimento completo da estrutura e uma compreensão de suas capacidades e limitações".

Na solução do problema clássico de elasticidade linear (análise matricial), tem-se: (a) determinação da matriz de rigidez de cada elemento através da aplicação do princípio da estacionaridade ao funcional da energia potencial total, no qual os deslocamentos dos pontos nodais são expressos em termos de funções aproximadoras escritas como funções de parâmetros incógnitos (aproximação nodal), (b) imposição das condições de contorno do problema aos deslocamentos o que significa a indicação de como esses deslocamentos se relacionam em cada ponto nodal (extremidade do elemento no caso de um elemento de barra com dois pontos nodais), e (c) determinação dos vetores de carga e indicação de como eles se somam nos nós. A solução do problema consiste em montar o sistema global de equações para a estrutura considerando-se para isto a contribuição em um ponto nodal de cada elemento a ele concorrente, resolver tal sistema determinando-se os deslocamentos e, a partir destes, encontrar os esforços em cada elemento bem como computar as reações de apoio.

Seguindo-se a filosofia da Programação Orientada a Objetos, tem-se grandes vantagens no uso da propriedade de herança quando aplicada à análise de estruturas como, por exemplo, no caso de análise elástica geometricamente não-linear uma sub-classe de uma classe **barras** seria criada para levar em consideração os efeitos de segunda ordem. No caso de conexões semi-rígidas, sub-classes da classe **nós**, ou mesmo da classe **barras**, seriam criadas. Uma lista de objetos de uma classe **modelos** poderia ser criada para simular o interrelacionamento entre as diversas fases do projeto no caso de análise de um processo evolutivo (sub-estruturação). As vantagens da propriedade da herança foram muito utilizadas nas classes **TElement**, **TNode**, **TLoad** e suas classes derivadas, as quais serão detalhadas nos Capítulos 5 e 6.

Neste projeto optou-se pela adoção da linguagem de programação C++ devido a esta oferecer as vantagens da orientação por objetos aliada a estruturação da linguagem tradicional (ou procedural). Outra vantagem é a existência de vários grupos de pesquisa que estão desenvolvendo seus trabalhos também nesta linguagem, o que facilita a comunicação, a utilização de códigos e a transferência de experiências. O ambiente desenvolvido foi compilado com o compilador Borland C++.

Em princípio o sistema não se destina a um determinado material, podendo ser utilizado tanto em estruturas de aço quanto em estruturas de concreto ou outro material. Também não inclui a geração do modelo (malha) que severá ser criado manualmente ou com a ajuda de *software* de terceiros.

A diversificação de sistemas operacionais e gráficos tornou-se um obstáculo à disseminação de programas, conforme salienta McGUIRE[1992]. Neste trabalho procurou-se a portabilidade do sistema, de modo que possa ser utilizado em computadores pessoais (PCs) ou estações de trabalho de alto desempenho, sob o sistema operacional UNIX, DOS e OS/2 ou, ainda, sob o ambiente WINDOWS. Nesse sentido o material apresentado por WEISKAMP et all [1993] (gerenciador de telas de texto e gráficas, ferramentas de tela, aplicações com o mouse, etc) se tornará de grande valia caso haja interesse no desenvolvimento de uma futura versão de pré e pós-processador apenas para PC rodando em ambiente DOS, enquanto que o material proposto por PETZOLD[1993] serve ao mesmo propósito para execução em ambiente WINDOWS.

O capítulo 2 descreve os conceitos necessários para a utilização da filosofia da linguagem orientada por objetos. O capítulo 3 apresenta o método dos elementos finitos e mostra a formulação geral para solução dos problemas através dessa técnica e o capítulo 4 a aplicação dessa técnica a um elemento de viga (barra) no espaço tridimensional. Nos capítulos 5 e 6 são apresentadas as classes já desenvolvidas e que compõem o sistema aqui proposto enquanto que o capítulo 7 descreve exemplos de estruturas aporticadas calculadas com este ambiente e compara os resultados obtidos com os gerados utilizando-se o programa **SAP-90** para realizar essas mesmas análises. Ao final, tem-se o capítulo 8 com as conclusões, o Apêndice A que traz um roteiro para a preparação do arquivo de entrada de dados deste programa.

2 - A PROGRAMAÇÃO ORIENTADA POR OBJETOS - POO

Devido aos avanços na capacidade de memória e rapidez de execução dos cálculos, bem como na redução dos preços dos computadores, a utilização destes sofreu um impulso nos últimos anos. Esse crescimento, aliado à exigência da parte dos usuários por programas mais complexos e com recursos cada vez maiores, trouxeram como consequência um desenvolvimento do campo das linguagens de programação. O maior desses desenvolvimentos é a filosofia da programação orientada a objetos, a qual designa um conceito diferente de programação unindo dados e sub-rotinas.

As linguagens tradicionais como **FORTRAN** (nome derivado de "*FORmula TRANslation*"), linguagem muito utilizada no desenvolvimento de programas que utilizam a técnica dos elementos finitos - FEM -, **C** ou **PASCAL** (estas com conceito de ponteiro, alocação dinâmica de memória, tipo de dados declarados e modularização) interpretam os dados como entidades passivas que podem ser manipuladas por "*procedures*" ou "*functions*" a partir de qualquer parte do código (SCHOLZ[1992]).

A programação orientada por objetos (OOP) é diferente da programação procedural tradicional porque nela o programa é formado por um grupo de objetos que se comunicam através do envio de mensagens. Essas mensagens acionam sub-rotinas, chamadas métodos, as quais agem sobre os dados do objeto que enviou a mensagem. Devido ser a mensagem a única possibilidade de se acessar os dados do objeto, esse processo fornece vantagens em relação a programação procedural, tais como a menor necessidade de manutenção dos programas e a maior facilidade de realização dessa manutenção quando ela se tornar necessária.

Na programação procedural dados e subrotinas são mantidos separados, unindo-se apenas quando aqueles são passados para estas nas suas chamadas. Na linguagem **FORTRAN** procedural o conceito de agrupamento de dados em um única estrutura inexiste, sendo necessário declarar várias variáveis separadamente e tomar o cuidado de mantê-las unidas. Na evolução das linguagens surgiu o conceito de estruturação de dados em um único tipo de dado (*record* para **PASCAL** e *struct* para **C**) permitindo que um conjunto de dados seja definido dentro de uma variável e mantenha-se nela sem fazer com que o programador se preocupe com isso. Evoluindo mais, a orientação a objetos utiliza-se do mesmo conceito de estruturação de dados dentro de um único tipo e o amplia com a inclusão de sub-rotinas mantendo-os, dados e sub-rotinas, unidos de modo a transformar esse conceito no conceito de classes.

2.1 - COMO INTERPRETAR A OOP

Fazendo-se um paralelo entre **OOP** e a linguagem **PASCAL** procedural podemos dizer que nesta montamos estruturas de dados (*records*) e, posteriormente, estabelecemos rotinas e funções (*procedures e functions*) que utilizam esses dados para executar tarefas, enquanto que naquela, os dados e as rotinas que sobre eles atuam permanecem unidos em uma só estrutura : a classe.

A programação procedural permite que os dados sejam manipulados de qualquer parte do programa por qualquer rotina que venha a ser implementada, facilitando a ocorrência de erros, dificultando a manutenção dos programas e a utilização de trechos do seu código no desenvolvimento do programa por terceiros ou em um outro projeto que seja semelhante.

Na **OOP** esses dados, rotinas e funções são combinados formando uma classe. Assim, um objeto - elemento representativo de sua classe -

possui os dados e métodos (rotinas ou funções), combinando-os com a finalidade de executar aquilo para que foi criado. Esses dados ficam encapsulados de modo que apenas os métodos do objeto conseguem acessá-los, ou seja, um objeto opera sobre seus próprios dados através de seus próprios métodos, de modo que funções externas à classe do objeto não consigam manipular suas variáveis.

Podemos observar na Figura 2.1 que a diferença entre a OOP e a programação estrutural tradicional está na organização e maneira de agrupar os códigos e os dados.

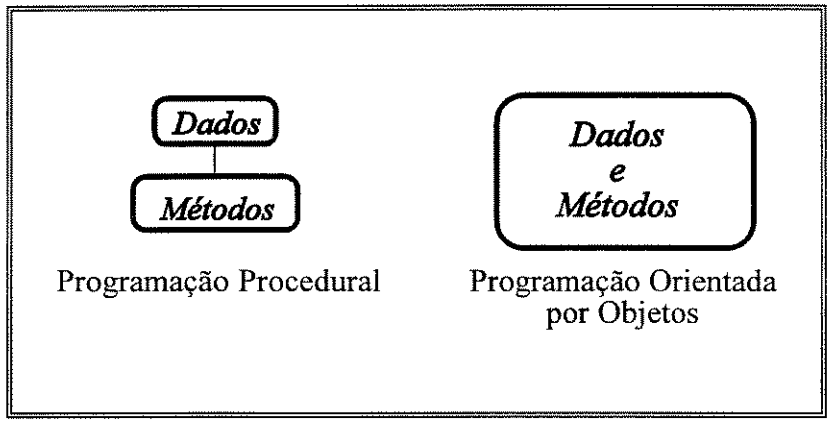

Figura 2.1 - Diferença entre OOP e Programação Tradicional

Temos então as seguintes diferenças básicas:

1 - Na programação tradicional (ou procedural) as linhas do código são projetadas em torno das funções. Assim sendo, estas são mais importantes neste tipo de programação que na OOP, na qual os objetos são mais importantes por serem o núcleo em torno do qual aglutinam-se os códigos. Dessa forma, em vez de se passar um objeto para a função como é feito na linguagem procedural, ele é utilizado para chamá-la, de modo que ela fique vinculada a ele.

2 - Na OOP pode-se evitar grandes funções que operem sobre casos de múltiplas possibilidades, optando-se por múltiplas classes, normalmente com

pequenas funções, que representem os diferentes componentes lógicos de um programa.

2.2 - ELEMENTOS DA OOP

2.2.1 - CLASSES

Segundo PAPPAS & MURRAY[1991], define-se classe como um novo tipo de dado, definido pelo usuário, que traz as vantagens da estruturação de dados em uma variável e a propriedade de limitar o acesso a esses dados específicos a funções próprias. Tais funções são, também, membros da classe, e ficam associadas aos objetos da classe. Sendo assim, as classes definem as propriedades e os atributos que descrevem as ações de um objeto, servindo, portanto, como um padrão para a criação de objetos que são também chamados de instâncias da classe.

Os componentes de uma classe são as variáveis (dados), que estão normalmente localizados numa área de acesso proibido para funções fora da classe, denominada área *protected* ou *private* (conforme o caso), e os métodos (funções), que definem o comportamento para um objeto, isto é, as ações que o objeto pode executar. Essas funções estão localizadas numa área denominada *public*, podendo ser acessadas de qualquer parte do programa, porém sempre associadas a um objeto da classe. Pode ocorrer de estarem localizadas na área *protected* ou *private*, sendo nesse caso chamadas por uma outra função pertencente a própria classe.

Na declaração de uma classe começa-se com com a definição de uma estrutura de dados na área *protected* ou *private* (que pode incluir tipos de dados declarados pelo usuário e até mesmo objetos de outras classes), assim como uma *struct* em **C** ou um *record* em **PASCAL**, e prossegue-se com a declaração dos métodos na área *public*.

Entre os métodos, geralmente encontram-se dois em especial : uma função construtora, chamada automaticamente toda vez que um objeto da classe for declarado e tem a finalidade de inicializá-lo; e uma função destrutora de objetos, que será chamada toda vez que se remover um objeto, retornando ao sistema a quantidade de memória por ele ocupada.

Pode-se implementar os métodos após sua declaração no mesmo arquivo ou em um arquivo separado e até mesmo implementá-lo na mesma linha de sua declaração. No caso da implementação dos métodos em arquivo separado daquele que contém sua declaração, o primeiro arquivo é dito arquivo de cabeçalho. Se a implementação se der na mesma linha (chamada implementação *"in line"*), esta deverá ser pequena, caso contrário o compilador a tratará como uma implementação normal.

A implementação *"in line"* tem importância relevante no tempo de execução de um programa. Segundo SWAN[1993], ao encontrar uma implementação desse tipo, o compilador insere o corpo da função diretamente no programa em vez de chamá-la como faria com uma função implementada de outra forma. Assim, inexistindo o desperdício de tempo que há com a chamada da função - chamados "*overheads*" -, diminue-se o tempo total gasto para executar o programa. Torna-se mais evidente essa vantagem se imaginarmos sua utilização dentro de um *loop* **for**, **while** ou outro qualquer, onde a chamada seria executada várias vezes até o término do procedimento.

2.2.2 - OBJETOS

Imaginando-se uma classe como se fosse um conjunto de elementos que contém aspectos comuns, um objeto, em poucas palavras, seria um elemento desse conjunto. Em uma visão mais técnica, objetos são extensões modernizadas do conceito de registro (*records* do **PASCAL** ou *struct* do **C**), que permitem organizar os dados em conjunto com as funções que trabalham com esses dados, unindo-os.

Como podem conter tanto dados como métodos, os objetos se parecem com miniprogramas, permitindo que possam ser usados para criar objetos mais complexos; como diz WEISKAMP et all [1993] "...muito parecido com o uso dado a transistores e relés na montagem de um circuito integrado...".

2.2.3 - MÉTODOS

Podem ser encarados como as rotinas ou funções declaradas em uma classe e que atua sobre os dados dos objetos dessa classe. São as mensagens passadas pelo objeto à sua classe que encontram, entre os métodos nela declarados e implementados, aquele que está associado a mensagem enviada. Assim, o objeto "responde" à mensagem com o seu método.

A mesma mensagem pode ser enviada a objetos diferentes que devido seus métodos próprios para essa mensagem, responderão de maneira diferente, determinando um comportamento denominado polimorfismo. Tal comportamento é de grande importância e muito explorado no desenvolvimento deste trabalho.

Os conceitos de objetos, classes e métodos originam os conceitos de abstração, modularidade e encapsulamento. Abstração é a capacidade de

representar entidades como objetos abstratos, suportando sua modelagem de maneira natural. Modularidade significa poder manipular a representação interna de um objeto apenas pelos seus próprios métodos, da sua maneira. Encapsulamento garante que um objeto responde apenas a seus métodos (WATSON & CHAN[1991]).

2.3 - PROPRIEDADES DA OOP

Tem-se duas propriedades básicas na **OOP** :

1 - Encapsulamento : é a propriedade que permite combinar os dados com as operações necessárias para processá-los (métodos) sob um teto de modo que aqueles somente sejam acessados e manipulados por estas. As operações e os dados estarão sempre associados a um objeto da classe. É o encapsulamento que dá aos objetos a aparência de blocos de construção. Sempre que houver uma classe esta propriedade se fará presente.

2 - Herança : é a propriedade que permite estender as classes que já se possui para abranger novos objetos formando sub-classes. Nesse caso a super-classe, também denominada classe raiz (termo mais apropriado para a primeira classe da qual as demais derivam direta ou indiretamente), contém dados e métodos que serão a base das outras classes derivadas, as quais, além desses, podem possuir seus próprios dados e métodos. Pode existir um programa completo, com várias classes, sem que se apresente a utilização desta propriedade, muito embora não seja provável tal acontecimento visto que várias são as vantagens da sua utilização.

2.3.1 - ENCAPSULAMENTO

O encapsulamento apresenta dois papéis importantes:

- Coloca dados sob um mesmo teto e
- Permite o ocultamento de dados.

Geralmente o encapsulamento tem três finalidades: proteger dados da exposição excessiva, ocultar os detalhes dos dados armazenados ou implementados (segundo WEISKAMP et all [1993] :" Não me interessa saber como é feito. Somente interessa-me que seja feito...") e facilitar o uso em um outro projeto de código já escrito e testado. Assim, o programador nunca precisa acessar diretamente os campos de dados de um objeto, bastando apenas utilizar seus métodos.

2.3.2 - HERANÇA

A herança permite obter uma nova classe de outra já existente denominando-se a classe obtida de subclasse ou classe derivada. Pode-se incluir dados e códigos (métodos) em uma subclasse sem ter que alterar a classe original, usar um código novamente ou modificar o comportamento da subclasse reescrevendo o código de alguns de seus métodos. Um objeto pode então herdar dados e métodos de uma classe, evitando-se assim a codificação das mesmas linhas de programa para tratar dois objetos de classes diferentes, porém semelhantes.

A herança permite também que um novo aplicativo seja criado com base nas classes de um outro existente sem, contudo, modificá-las. Isso é conseguido apenas derivando-se subclasses que conterão todas as modificações necessárias e traz, entre outras, as vantagens de não se correr o risco de introduzir erros no aplicativo original e, também, de não ter que testá-lo (o original) ao término das modificações.

Pode-se criar uma hierarquia de classes onde há uma classe-raiz com tarefas mais gerais ou até mesmo abstrata, da qual não se pode declarar nenhum objeto, e várias subclasses ou classes derivadas representando tarefas mais específicas, fazendo com que objetos tenham comportamento diferentes, mesmo tendo a mesma origem, o que caracteriza o polimorfismo. Tal efeito é conseguido fazendo-se os métodos herdados similares (com detalhes diferentes na implementação) e respondendo a uma mesma mensagem. Vários artigos exemplificam o uso da propriedade da herança (WATSON and CHAN[1991], MACKIE[1992] , SCHOLZ[1992]).

Nos problemas de elementos finitos aplicados ao estudo de estruturas, notam-se comportamentos comuns nos diversos elementos (composição por nós; são feitos de algum tipo de material; a existência de nós inicial e final em uma barra; etc), que podem ser herdados sem modificação pelos diversos objetos que discretizam o problema. Os comportamentos semelhantes, porém diferentes, como a obtenção da matriz de rigidez de uma barra ou de uma placa ou mesmo daquela levando em consideração a não linearidade geométrica, podem herdar o mesmo método para obtê-la, porém devem conter diferentes implementações de cálculo. Deste modo, a implementação de uma função que obtém a matriz global requer apenas a chamada a função que calcula as diversas matrizes dos elementos, sem se preocupar como são efetuados esses cálculos.

Das características da filosofia de programação orientada por objetos expostas acima, conclui-se que tal paradigma é útil na criação de programas que serão facilmente modificados se houver necessidade, pois são compostos por funções de simples reutilização, bem como de programas que possam se valer de uma parte ou mesmo da totalidade de um outro, servindo este como uma plataforma de apoio. Desta maneira agiliza-se o trabalho dos programadores, que além do tempo e da certeza de não incluir erros em código já testado e considerado estável, ganham maior flexibilização quando declaram uma classe.

3 - ELEMENTOS FINITOS - FORMULAÇÃO

3.1 - APRESENTAÇÃO

Projetos mais complexos e/ou sujeitos a mais restrições de segurança têm sido requisitados pela evolução da tecnologia (projetos espaciais, nucleares, de fluxo de calor ou hidráulico, poluição, etc). Para o estudo desses projetos há necessidade de modelá-los (representá-los matematicamente) de modo a permitir a simulação do comportamento do sistema fisicamente complexo. Ciências como a mecânica dos sólidos permitem descrever o comportamento desses sistemas físicos.

Os problemas que possam ser representados por um número finito de componentes são ditos discretos e podem ser solucionados por computador, enquanto aqueles que são definidos usando-se elementos matemáticos infinitesimais chamam-se contínuos e apenas podem ser resolvidos através de manipulações matemáticas que conduzem a equações diferenciais as quais nem sempre apresentam solução.

Para se solucionar os problemas contínuos utilizam-se vários métodos de discretização, os quais envolvem aproximações, sendo um desses métodos (técnica) o dos elementos finitos. Basicamente o método consiste em transformar as equações derivadas parciais em equações algébricas utilizando uma transformação (aproximação) simples para as variáveis incógnitas.

Esse método é largamente aplicado nos problemas de mecânica estrutural, transferência de calor e mecânica dos fluidos (problemas lineares ou não, estacionários ou não, definidos em um domínio geométrico de qualquer dimensão), podendo ser aplicado também a qualquer outro problema que se traduza na solução de um conjunto de equações diferenciais.

Considere-se um corpo tridimensional de volume **V** e superfície **S** tendo qualquer um de seus pontos sujeito a um estado de tensões e de deformações dados por:

$$\{\sigma\} = \left\{\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{yz}, \sigma_{zx}\right\}$$

$$\{\varepsilon\} = \left\{\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \varepsilon_{xy}, \varepsilon_{yz}, \varepsilon_{zx}\right\}$$

Atuam também sobre um ponto do volume **V** do corpo as forças volumétricas

$$\{F\} = \left\{F_x, F_y, F_z\right\}$$

e, em um ponto na superfície **S**, as cargas aplicadas

$$\{P\} = \left\{P_x, P_y, P_z\right\}$$

Sejam os movimentos de qualquer ponto do corpo descritos pelas componentes do vetor deslocamento

$$\{u\} = \left\{u_x, u_y, u_z\right\}$$

Para encontrar o funcional que representa a energia potencial total desse corpo deve-se somar as parcelas da energia de deformação **U** (numericamente igual ao trabalho realizado pelos esforços internos) e a parcela Ω do trabalho realizado pelas ações (forças volumétricas {**F**} e ações externas {**P**}). Sendo assim, tendo-se

$$= \frac{1}{2}\int_V \{\varepsilon\}^T \{\sigma\}\, dV \qquad (3.1)$$

e

$$\Omega = -\int_V \{u\}^T \{F\}\, dV - \int_S \{u\}^T \{P\}\, dS \qquad (3.2)$$

Resulta da soma das equações 3.1 e 3.2 a expressão para a energia potencial total que é dada pela equação 3.3 à seguir

$$\Pi = U + \Omega = \frac{1}{2}\int_V \{\varepsilon\}^T \{\sigma\}\ dV - \int_V \{u\}^T \{F\}\ dV - \int_S \{u\}^T \{P\}\ dS \qquad (3.3)$$

onde Π representa o funcional da energia potencial total no domínio.

Os sinais presente nas expressões da energia de deformação (eq. 3.1) e da energia potencial das cargas (eq. 3.2), segundo TIMOSHENKO e GERE[1984], resultam de ser a energia potencial de um sistema mecânico igual ao trabalho realizado por todas as forças atuantes quando a estrutura é movida da configuração deslocada (com carga) à configuração de referência (sem carregamento). Nesse movimento da estrutura, para as forças internas (que para vigas, treliças e pórticos são as tensões resultantes), a quantidade de trabalho recuperado é igual à energia de deformação, daí o sinal positivo, enquanto para o caso das cargas externas (forças volumétricas e ações externas) a energia potencial recebe sinal negativo porque cada carga executa trabalho negativo.

TIMOSHENKO e GERE[1984] ressaltam, também, que a energia potencial de uma carga **P,** qualquer, é diferente do trabalho realizado por essa carga durante o carregamento. No caso do carregamento, a carga tem sua magnituda aumentada, gradativamente, de zero ao seu valor final **P**, enquanto a energia potencial é numericamente igual ao trabalho realizado pela força (com sua magnitude plena) ao se mover a estrutura da posição real para a de referência.

O princípio da energia potencial estacionária (também chamado de princípio de Kirchhoff), ainda segundo TIMOSHENKO e GERE[1984], estabelece que, sendo a energia potencial de uma estrutura elástica (linear ou não-linear) expressa em função dos deslocamentos das juntas, a estrutura estará em equilíbrio quando esses deslocamentos forem tais que levem a energia potencial total a atingir um valor estacionário. Estando a estrutura em equilíbrio estável, a energia potencial total

assumirá um valor mínimo, e caso o equilíbrio seja instável, a energia potencial total tanto pode assumir um valor máximo como neutro.

Deve-se salientar que o princípio acima conduz às equações de equilíbrio apenas para sistemas com relações tensão x deformação linear, pois para sistemas não-lineares podem haver mais de uma configuração de equilíbrio (ASSAN[1993]).

O princípio variacional estabelece que sendo um funcional (escalar) dado pela forma

$$\bar{\Pi} = \int_{\Psi} F\left(u, \frac{d}{dx}u, \frac{d^2}{dx^2}u \ldots, \frac{d}{dy}u, \frac{d^2}{dy^2}u, \ldots\right) d\Psi + \int_{\Sigma} E\left(u, \frac{d}{dx}u, \frac{d^2}{dx^2}u \ldots, \frac{d}{dy}u, \frac{d^2}{dy^2}u, \ldots\right) d\Sigma \qquad (3.4)$$

no qual $\boldsymbol{u}$ é uma função incógnita e $\boldsymbol{F}$ e $\boldsymbol{E}$ são operadores específicos, estabelece-se a solução para o problema do contínuo encontrando-se a função $\boldsymbol{u}$ que torna o funcional estacionário com respeito a pequenas variações $\delta\boldsymbol{u}$, ou seja, deve-se ter

$$\delta\ \bar{\Pi}\ =\ 0 \qquad (3.5)$$

Agora, minimizando o funcional dado pela equação 3.3 como indicado na equação 3.5 (ou seja, estabelecendo o ponto estacionário), tem-se:

$$\delta\ \Pi = \int_V \{\delta\,\varepsilon\ \}^T\ \{\sigma\}\ dV - \int_V \{\delta\ u\ \}^T\ \{F\}\ dV - \int_S \{\delta\ u\ \}^T\ \{P\}\ dS = 0 \qquad (3.6)$$

Da equação 3.6 conclui-se que

$$\int \{\delta\,\varepsilon\ \}^T\ \{\sigma\}\ dV = \int_V \{\delta\ u\ \}^T\ \{F\}\ dV + \int_S \{\delta\ u\ \}^T\ \{P\}\ dS \qquad (3.7).$$

A equação acima (3.7) representa a igualdade dos trabalhos virtuais interno e externo, e qualquer um de seus membros representa um conjunto de equações que equivalem ao equilíbrio do elemento.

Na técnica dos elementos finitos (aqui seguindo-se o desenvolvimento proposto por BREBBIA & FERRANTE[1975] e por ASSAN[1993]), o funcional da energia potencial total é substituido por um funcional aproximado ($\Pi_{\mathbf{a}}$), no qual as variáveis do problema são colocadas em termos de combinação linear de funções de interpolação - as funções de forma ("*shape functions*") - multiplicadas por parâmetros a determinar.

Os parâmetros incógnitos referidos no parágrafo anterior são os valores das variáveis do problema nos pontos nodais do elemento (no contorno ou dentro dele). Assim o valor do funcional aproximado da energia potencial total para todo o domínio será a soma dos funcionais aproximados (que exprimem a energia potencial total) de cada elemento, ou seja

$$\Pi_a = \sum_{i=1}^{m} \Pi_{a\,i} \qquad (3.8)$$

onde

m = número de elementos em que foi dividido o domínio

$\Pi_{a\,i}$ = funcional aproximado para o elemento i;

Sendo as funções de interpolação η_j conhecidas (funções de forma) e os parâmetros c_j incógnitos para um elemento **i**, pode-se escrever que a função aproximadora dos deslocamentos de cada elemento será

$$\mathrm{v} = \sum_{j=1}^{n} c_j\, \eta_j \qquad (3.9)$$

Assim, o funcional aproximado do domínio fica sendo expresso por

$$\Pi_a\left(c_j\right) = \sum_{i=1}^{m} \Pi_{a\,i}\left(c_j\right) \qquad (3.10)$$

Aplicando-se à equação 3.10 a condição de estacionaridade do funcional colocada na equação 3.5 (conceito da variação do funcional nula), tem-se

$$\delta \Pi_a \left(c_j\right) = \sum_{j=1}^{n} \delta \Pi_{a_i} \left(c_j\right) = 0 \qquad (3.11)$$

ou seja

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \frac{\partial \Pi_{a_i}\left(c_j\right)}{\partial c_j} = 0 \qquad (3.12)$$

Esta equação representa um conjunto de equações algébricas lineares simultâneas cuja solução determina os valores dos parâmetros c_j que são os valores das variáveis do problema nos pontos nodais. Deve-se reafirmar que a equação 3.5 representa a solução exata para o problema, enquanto a equação 3.12 fornece uma solução aproximada.

Resumindo-se, a técnica consiste em dividir o domínio contínuo em um número finito de elementos (dai o nome elementos finitos) , formando uma rede denominada malha de elementos finitos, unidos pelos pontos nodais, e admitir funções individuais para cada elemento do domínio de tal forma que cada uma delas satisfaça as condições de contorno do seu elemento correspondente.

Dessa forma os problemas existentes em outros métodos de solução, tais como:

- a dificuldade de se obter uma única função que satisfaça as condições de contorno irregular do domínio como um todo;

- saber se essa função adotada se aproxima da função exata (para haver maior precisão no cálculo);

- ou a admissão de funções de ordem superior para minorar o erro tornando o cálculo muito trabalhoso e, às vezes, impraticável, são contornados.

O método dos elementos finitos leva à obtenção de uma solução que tende ao valor exato da maneira mostrada na figura 3.1 a seguir. Como se percebe, tanto maior será a precisão dos resultados quanto maior for o número de elementos.

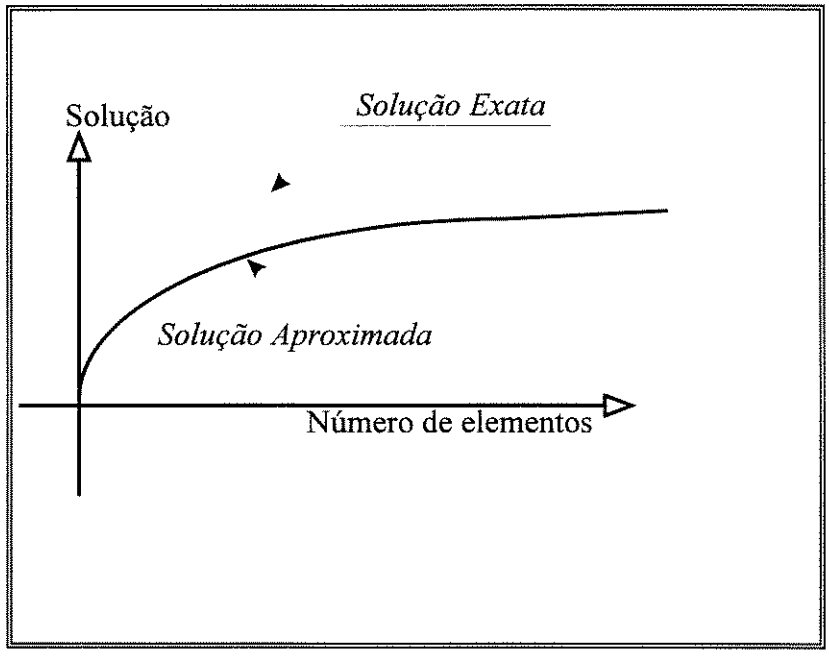

Figura 3.1 - Convergência (neste caso "por baixo") da solução aproximada à solução exata

Devido à modularização do método, torna-se relativamente fácil sua implementação em programas que seguem a filosofia da orientação por objetos, tendo como passos principais os relacionados a seguir (MACKIE[1992]), que são a base de qualquer programa de elementos finitos:

1. entrada de dados;
2. cálculo das matrizes de rigidez e dos vetores de carga individuais;
3. montagem da matriz de rigidez global e do vetor de cargas global;
4. imposição das restrições (condições de contorno do problema);
5. solução do sistema de equações;
6. apresentação dos resultados.

3.2 - MODELO DE FORMULAÇÃO ADOTADO

De acordo com BREBBIA & FERRANTE[1975], na mecânica dos sólidos existem quatro alternativas para estabelecer o modelo de elemento finito (que é a maneira pela qual o elemento finito é formulado), sendo que tais alternativas diferem no princípio variacional adotado e no tipo de comportamento assumido para cada elemento.

- **MODELO COMPATÍVEL** - Conhecido como método dos deslocamentos ou da rigidez. É derivado do princípio da mínima energia potencial. Assume-se um conjunto de deslocamento para cada elemento que preserve as condições de compatibilidade dos deslocamentos entre os elementos. As incógnitas são os deslocamentos nodais.

- Os três modelos restantes são : **MODELO DO EQUILÍBRIO** (conhecido como método das forças, é derivado do princípio da mínima energia complementar), **MODELO HÍBRIDO** e **MODELO MISTO**.

Neste trabalho é utilizado o modelo compatível, desenvolvendo-se a seguir a formulação geral para o método.

3.3 - FORMULAÇÃO

A hipótese básica da formulação diz que os deslocamentos para qualquer ponto do elemento podem ser representados por funções aproximadoras escritas em termos de parâmetros incógnitos (BREBBIA & FERRANTE[1975]). Assim, particularizando-se para uma viga, por exemplo, se os deslocamentos nas direções x, y e z forem dados respectivamente pelas equações (funções aproximadoras) seguintes

$$u_x = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

$$u_y = a_4 + a_5 x + a_6 x^2 + a_7 x^3$$

$$u_z = a_8 + a_9 x + a_{10} x^2 + a_{11} x^3,$$

e sendo

$$\{u\}^T = \{u_x, u_y, u_z\},$$

conforme definido acima, pode-se dizer que, em forma matricial,

$$\{u\} = [N]\ \{a\} \qquad (3.13)$$ onde

$\{u\}$ = vetor que contém as funções aproximadoras do elemento

$[N]$ = matriz função da posição (coordenadas)

$\{a\}$ = vetor dos parâmetros generalizados

Para as funções deslocamentos adotadas, tem-se que a matriz $[N]$ será

$$[N]=\begin{bmatrix} 1 & x & x^2 & x^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & x & x^2 & x^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x & x^2 & x^3 \end{bmatrix}$$

e o vetor dos parâmetros generalizados

$$\{a\}^T = \{a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7 \quad a_8 \quad a_9 \quad a_{10} \quad a_{11}\}$$

Da equação 3.13 pode-se escrever que a variação dos deslocamentos será dado por

$$\delta \{u\} = [N] \delta \{a\} \qquad (3.14)$$

Sendo o vetor dos valores dos deslocamentos (ou incógnitas) nodais

$$\{v\}^T = \{v_1, v_2, v_3, \ldots, v_i\}$$

e colocando-se as funções aproximadoras em funções dessas incógnitas nodais - ressalte-se que nem todas as incógnitas são deslocamentos (ASSAN[1993]) -, resulta:

$$\{u\} = [G] \{v\} \qquad (3.15) \text{ onde}$$

$[G]$ é denominada matriz das funções de forma

Impondo-se as condições de contorno do elemento às incógnitas nodais, relaciona-se $\{v\}$ com os parâmetros generalizados $\{a\}$ da seguinte maneira:

$$\{v\} = [A] \{a\} \qquad (3.16)$$

Logo, se as funções adotadas forem adequadas, a matriz $[A]$ admite inversa, e pode-se equacionar

$$\{a\} = [A]^{-1} \{v\} \qquad (3.17)$$

de onde escreve-se que a variação das incógnitas nodais é dada por:

$$\delta \{a\} = [A]^{-1} \delta \{v\} \qquad (3.18).$$

Da mecânica dos sólidos tem-se as seguintes relações:

- Relação Deformação-Deslocamento ou Relações de Compatibilidade

$$\{\varepsilon\} = [L] \{u\} \qquad \text{eq. A.1}$$

- Relação Tensão-Deslocamento

$$\{\sigma\} = [C] [L] \{u\} \qquad \text{eq. A.2}$$

Substituindo as equações 3.13 e 3.17 na equação A.1, resulta:

$$\{\varepsilon\} = [L] [N] [A]^{-1} \{v\} \qquad (3.19)$$

ou seja, fazendo

$$[B] = [L] [N] [A]^{-1},$$

resulta

$$\{\varepsilon\} = [B] \{v\} \qquad (3.20)$$

Partindo-se da relação Tensão-Deslocamento (equação A.2) e subustituindo-se as relações encontradas nas equações 3.13 e 3.17 encontra-se:

$$\{\sigma\} = [C] [L] [N] [A]^{-1} \{v\} \qquad (3.21)$$

Substituindo-se o valor de $[B]$, tem-se

$$\{\sigma\} = [C] [B] \{v\} \qquad (3.22)$$

De posse das relações 3.20, 3.22 e lembrando que

$$\{u\} = [N] \{a\} = [N] [A]^{-1} \{v\}$$

pode-se completar a formulação do problema substituindo essas relações na equação3.7. Chega-se, então, a:

$$\{\delta v\}^T \int_V [B]^T \{\sigma\}\, dV = \{\delta v\}^T \int_V \left([N]\,[A]^{-1}\right)^T \{F\}\, dV + \{\delta v\}^T \int_S \left([N]\,[A]^{-1}\right)^T \{P\}\, dS$$

(3.23)

Como os incrementos de deslocamentos ($\{\delta v\}$) são arbitrários, a equação 3.23 resulta em:

$$\int_V [B]^T \{\sigma\}\, dV = \int_V \left([N]\,[A]^{-1}\right)^T \{F\}\, dV + \int_S \left([N]\,[A]^{-1}\right)^T \{P\}\, dS \qquad (3.24)$$

Substituindo a expressão 3.22 na relação 3.24, tem-se:

$$\int_V [B]^T [C]\,[B]\,\{v\}\, dV = \int_V \left([N]\,[A]^{-1}\right)^T \{F\}\, dV + \int_S \left([N]\,[A]^{-1}\right)^T \{P\}\, dS \qquad (3.25)$$

Reescrevendo-se a equação 3.25:

$$\left(\int_V [B]^T [C]\,[B]\, dV\right)\{v\} = \left([A]^{-1}\right)^T \left(\int_V [N]^T \{F\}\, dV + \int_S [N]^T \{P\}\, dS\right) \qquad (3.26)$$

Neste ponto atribui-se o nome de matriz de rigidez do elemento finito ao termo entre parêntesis do primeiro membro da relação 3.26, e representa-a por [k]. Assim, tem-se

$$[\mathrm{k}] = \left(\int_V [B]^T [C]\,[B]\, dV\right) \qquad (3.27);$$

e ao termo do segundo membro da equação 3.26 dá-se o nome de vetor de cargas nodais equivalentes e representa-o por {r}, ficando

$$\{r\} = \left([A]^{-1}\right)^T \left(\int_V [N]^T \{F\}\, dV + \int_S [N]^T \{P\}\, dS\right) \qquad (3.28)$$

A relação 3.26 fornece um conjunto de equações que solucionam o problema para um elemento. Estendendo-se o sistema para todo o domínio, cuja matriz de rigidez é $\mathbf{K}$, vetor de incógnitas nodais é $\mathbf{V}$ e vetor das cargas nodais equivalentes é $\mathbf{R}$, tem-se a seguinte relação:

$$KV = R \qquad (3.29)$$

A relação 3.29 permite solucionar o problema através da técnica dos elementos finitos, tratando-o como um todo transformado de contínuo em discreto. No capítulo seguinte será desenvolvida a formulação do método para elementos de barra de modo a poder ser feita a aplicação na análise de estruturas apórticadas espaciais.

4 - FORMULAÇÃO DO PROBLEMA VARIACIONAL PARA UM TRECHO GENÉRICO DE VIGA NO ESPAÇO TRIDIMENSIONAL E OBTENÇÃO DA MATRIZ DE RIGIDEZ

4.1 - FUNCIONAL DA ENERGIA POTENCIAL TOTAL

A formulação do método dos elementos finitos para um elemento de viga (barra) no espaço tridimensional será feita adotando-se, de acordo com a figura 4.1, o mesmo sistema de eixos que o proposto por GERE[1987], no qual **X**, **Y** e **Z** formam um sistema de eixos para a estrutura, chamado de sistema global de coordenadas, e $\mathbf{X_m}$,$\mathbf{Y_m}$ e $\mathbf{Z_m}$ formam um outro sistema de eixos para o elemento, denominado sistema local de coordenadas.

Fazendo-se o eixo $\mathbf{X_m}$ coincidir com o eixo longitudinal do elemento e o eixo $\mathbf{Z_m}$ ficar contido no plano formado pelos eixos **X** e **Z** da estrutura, tem-se que o eixo $\mathbf{Y_m}$ pertencerá ao plano que passa pelos eixos $\mathbf{X_m}$ e **Y**. O sistema local assim formado é obtido fazendo-se o sistema global rotacionar de um ângulo β em torno do eixo **Y**, com a posterior rotação em torno do eixo **Z** (agora na posição $\mathbf{Z_m}$) de um ângulo igual a γ (ver figura 4.1).

Considerando-se que o funcional que representa a energia potencial total é a soma das contribuições dos funcionais de energia potencial relativos aos esforços e deslocamentos normal, de flexão em dois planos e o de torção no plano restante, ou seja,

$$\Pi_{Total} = \Pi_{Normal} + \Pi_{Flexão\ XmYm} + \Pi_{Flexão\ XmZm} + \Pi_{Torção\ YmZm}$$

calcula-se, a seguir, a expressão da contribuição de cada uma dessas parcelas. Como essas parcelas são formadas pela energia de deformação e pelo trabalho realizado

pelas forças que atuam no elemento, necessitam que sejam adotados alguns tipos de carregamento (os mais comuns encontrados na prática) atuando sobre ele, elemento.

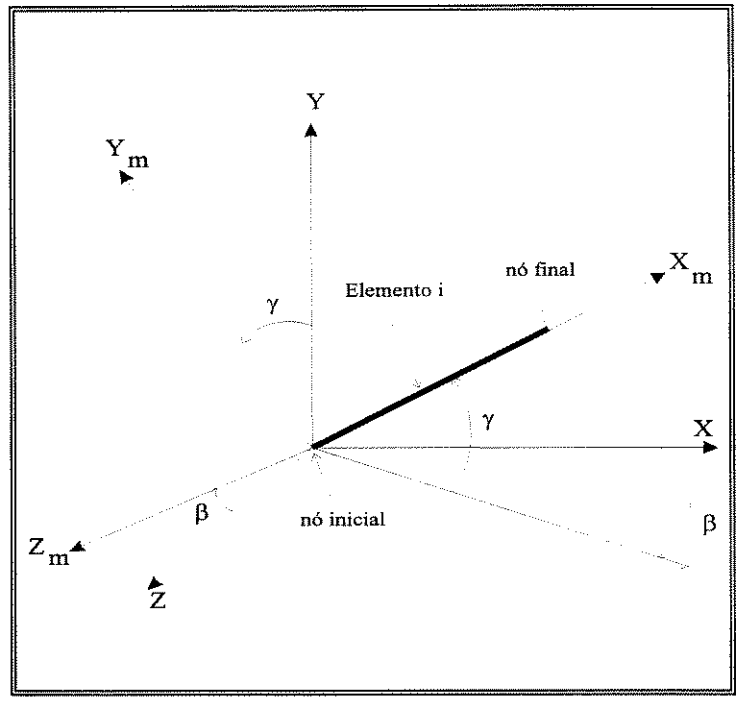

Figura 4.1 - Sistema de eixos do elemento de barra (Sistema Local de Coordenadas)

4.1.1 - PARCELA DEVIDA AO ESFORÇO E DESLOCAMENTO NORMAL

Seja um trecho genérico de uma viga, apresentado na figura 4.2, com condições de contorno (deslocamentos das extremidades) formando o vetor

$$\left\{\begin{matrix}-\\u\end{matrix}\right\}^T = \{u_1, u_7\}^1$$

e sujeita a atuação de cargas concentrado $\mathbf{P_1...P_i}$ e da carga linerarmente distribuida **N(x)**, ambas atuando ao longo do seu eixo longitudinal.

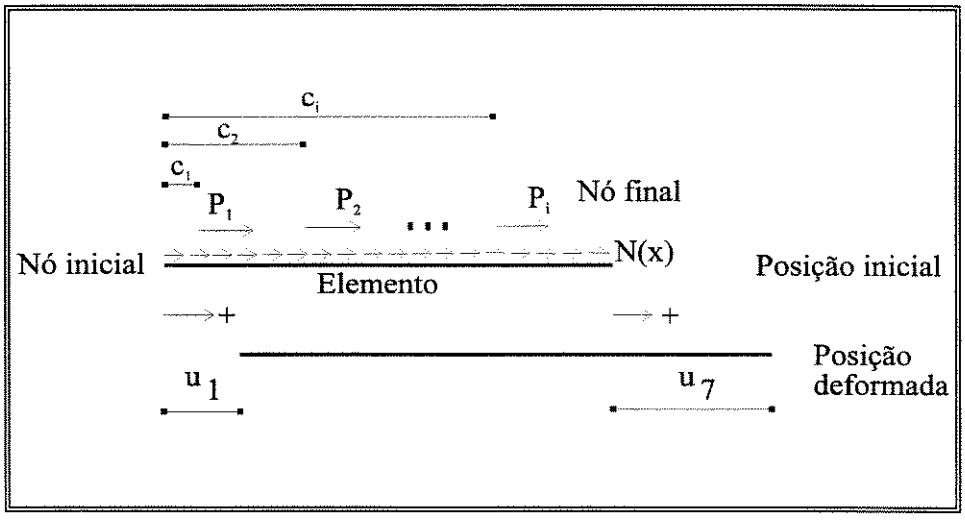

Figura 4.2 - Trecho de uma viga com deslocamentos u_1 e u_2 nas extremidades e cargas N(x), $P_1...P_i$

[1]Sabendo-se que existem seis graus de liberdade por nó, o número do índice dos deslocamentos correspondentes em nós consecutivos será acrescido de seis unidades por simplificação.

4.1.1.a) Função aproximadora

Para a obtenção da solução aproximada, o espaço vetorial das funções de deslocamentos possíveis é substituido pelo sub-espaço vetorial dos polinômios de 1º grau $\bar{u}(x)$, tal que

$$\bar{u}(x) = u_1\,\eta_1(x) + u_7\,\eta_7(x) \qquad \text{(eq. 4.1)}^2$$

onde $\eta_1(x)$ e $\eta_2(x)$ são funções lineares com as seguintes condições de contorno (de acordo com a figura 4.3) :

$$\bar{u}(0) = u_1 \Rightarrow \begin{cases} \eta_1(0) = 1 \\ \eta_2(0) = 0 \end{cases} \qquad \text{e} \qquad \bar{u}(L) = u_7 \Rightarrow \begin{cases} \eta_1(L) = 0 \\ \eta_2(L) = 1 \end{cases}$$

Para um ponto numa posição x qualquer temos, por semelhança de triângulos,

$$\left.\begin{array}{rl} & \rightarrow L \\ \eta_1(x) & \rightarrow L - x \end{array}\right\} \Rightarrow \eta_1(x) = \frac{L-x}{L}$$

e

$$\left.\begin{array}{rl} & \rightarrow L \\ \eta_7(x) & \rightarrow x \end{array}\right\} \Rightarrow \eta_7(x) = \frac{x}{L}$$

Logo, a função aproximadora (eq. 4.1) assume a forma

$$\bar{u}(x) = u_1\left(1 - \frac{x}{L}\right) + u_7\left(\frac{x}{L}\right) \qquad \text{(eq. 4.2)}$$

Nota-se que a derivada da expressão 4.2 em relação à x resulta em

$$\frac{d}{dx}\left(\bar{u}(x)\right) = \frac{u_7 - u_1}{L} \qquad \text{(eq. 4.3)}$$

que representa a variação do deslocamento por unidade de comprimento (quantidade final menos a quantidade inicial, dividido pelo comprimento) .

---

[2]Essa aproximação é conhecida como aproximação nodal.

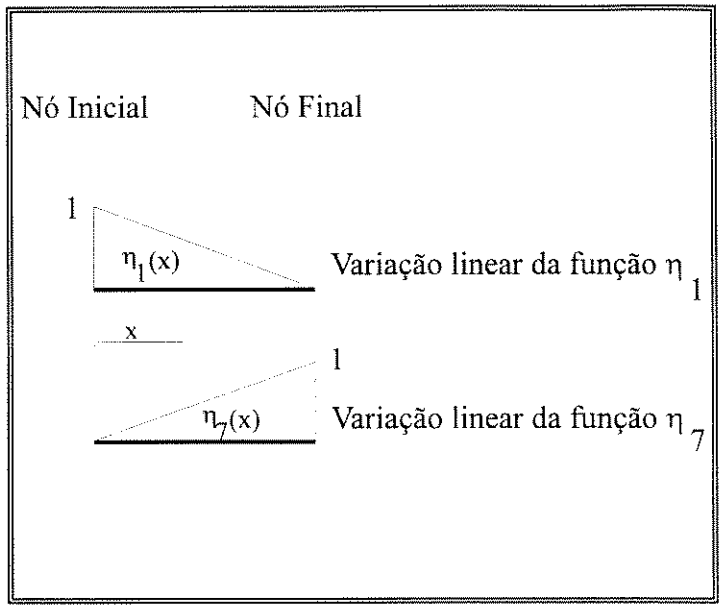

Figura 4.3 - Esquema de variação linear para as funções η_1 e η_7

4.1.1.b) Expressão da Energia de Deformação

Considerando-se a expressão da energia de deformação

$$U = \int_V \frac{1}{2} \sigma \varepsilon \, dV \qquad \text{(eq. 4.4)}$$

Para uma distribuição constante das tensões e deformações ao longo da área do elemento, tem-se que

$$U = \int_0^L \frac{1}{2} \sigma \varepsilon \left(\int_A dA \right) dx = \int_0^L \frac{1}{2} A \, \sigma \varepsilon \, dx$$

Para um esforço normal

$$\sigma = \frac{N}{A}$$

onde **N** é a força normal atuante na seção.

Tem-se, também, que

$$E\varepsilon = \frac{N}{A},$$

de onde,

$$N = EA\varepsilon$$

Substituindo-se primeiramente a relação da tensão com a normal, e, posteriormente, a relação da normal com o módulo de elasticidade, a área e a deformação na expressão acima, encontra-se

$$U_{Normal} = \int_0^L \frac{1}{2} A \frac{N}{A} \varepsilon \, dx$$

$$U_{Normal} = \frac{1}{2} \int_0^L EA\varepsilon^2 \, dx$$

Comparando-se a deformação que ocorre neste caso com as possíveis numa análise tridimensional, chega-se a conclusão que a única existente é

$$\varepsilon = \varepsilon_{11} = \frac{\partial u_1}{\partial x_1},$$

ficando, com a substituição da função de deslocamento $u(x)$ pela função aproximadora $\bar{u}(x)$,

$$\varepsilon = \frac{d}{dx}\left(\bar{u}(x)\right)$$

Assim, considerando-se um elemento cuja seção transversal seja constante ao longo do comprimento

$$U_{Normal} = \frac{EA}{2} \int_0^L \left(\frac{d}{dx}\left(\bar{u}(x)\right)\right)^2 dx \qquad \text{(eq 4.5)}$$

Substituindo a equação 4.3 na equação 4.5, tem-se

$$U_{Normal} = \frac{EA}{2}\int_0^L \left(\frac{(u_7 - u_1)}{L}\right)^2 dx$$

Integrando-se, resulta a expressão da energia de deformação procurada, ou seja,

$$U_{Normal} = \frac{1}{2}\frac{EA}{L}\left(u_7{}^2 - 2u_1u_7 + u_1{}^2\right) \qquad \text{(eq. 4.6)}$$

4.1.1.c) Energia Potencial do Trabalho das Forças Externas

Considerando-se a expressão do trabalho das forças externas como força vezes deslocamento e, com o carregamento de várias forças concentradas $\mathbf{P_1...P_i}$ aplicadas respectivamente a uma distância $\mathbf{c_1...c_i}$ da origem do eixo, e uma força **N(x)** distribuída ao longo do eixo por todo o comprimento do elemento, tem-se que o trabalho total das forças externas é dado por

$$\Omega_{Normal} = -\sum_{j=1}^{i} P_j \ \bar{u}(c_j) \ - \int_0^L N \ \bar{u}(x)\, dx \qquad \text{(eq. 4.7)}^3$$

Assim, substituindo-se na equação 4.7 o valor da função aproximadora nos pontos de aplicação das cargas concentradas (valor da equação 4.2 nos pontos $x = \mathbf{c_i}$), tem-se

$$\Omega_{Normal} = -\sum_{j=1}^{i} P_j\left(u_1 + \frac{c_j}{L}(u_7 - u_1)\right) - \int_0^L N\left[u_1 + \left(\frac{u_7 - u_1}{L}\right)\right]dx \qquad \text{(eq. 4.8)}$$

Integrando-se a equação 4.8 encontra-se

$$\Omega_{Normal} = -\sum_{j=1}^{i} P_j \ u_1 - \sum_{j=1}^{i}\frac{P_j \ c_j}{L}(u_7 - u_1) - N L \, u_1 - \frac{N L}{2}(u_7 - u_1)$$

[3] As parcelas que compõem a expressão da energia potencial do trabalho das forças externas têm sinais negativos na formação da expressão da energia potencial total, conforme explicado no capítulo 3.

Reescrevendo-se, resulta

$$\Omega_{Normal} = \underbrace{-\sum_{j=1}^{i} P_j \left(\frac{L-c_j}{L}\right) u_1 - \sum_{j=1}^{i} P_j \frac{c_j}{L} u_7}_{\text{parcela relativa aos esforç osconcentrados}} \underbrace{-\frac{NL}{2} u_1 - \frac{NL}{2} u_7}_{\text{parcela relativa ao carregamento distribuido}} \qquad \text{(eq. 4.9)}$$

Rearranjando-se a equação 4.9 em termos das incógnitas u_1 e u_7 tem-se a expressão para o trabalho realizado pelo carregamento em questão

$$\Omega_{Normal} = \left\{-\left[\sum_{j=1}^{i} P_j \left(\frac{L-c_j}{L}\right)\right] - \frac{NL}{2}\right\} u_1 - \left[\left(\sum_{j=1}^{i} P_j \frac{c_j}{L}\right) + \frac{NL}{2}\right] u_7 \qquad \text{(eq. 4.10)}$$

4.1.2 - PARCELA DEVIDA AO ESFORÇO FLETOR E DESLOCAMENTOS CORRESPONDENTES

Para a consideração da contribuição dos esforços de flexão primeiramente estuda-se a flexão que ocorre no plano **XY** e depois, de posse da expressão encontrada para esse plano, encontra-se a expressão para o mesmo efeito atuando no plano **XZ**. Assim sendo, seja um trecho genérico de viga, apresentado na figura 4.4, com condições de contorno (deslocamentos dos extremos) formando o vetor

$$\left\{\begin{matrix} - \\ v \end{matrix}\right\}^T = \{v_2, \theta_3, v_8, \theta_9\}$$

e sujeita a atuação dos esforços concentrados $\mathbf{P_{f1}}...\mathbf{P_{fi}}$, uniformemente distribuído **Q** e linearmente distribuido **q(x)**, todos atuando perpendicularmente ao eixo longitudinal do elemento (barra, no caso).

4.1.2.a) Função aproximadora

Para a obtenção da solução aproximada, o espaço vetorial das funções de deslocamentos possíveis é substituido pelo sub-espaço vetorial dos polinômios de 3º grau $\bar{v}(x)$, tais que

$$\bar{v}(x) = v_2\ \eta_2(x) + \theta_3\ \eta_3(x) + v_8\ \eta_8(x) + \theta_9\ \eta_9(x) \qquad \text{(eq. 4.11)}$$

Dessa forma, para um ponto numa posição x qualquer o valor aproximado do seu deslocamento será dado pela combinação dos valores das curvas da figura 4.5 nesse ponto seguindo-se a equação 4.11.

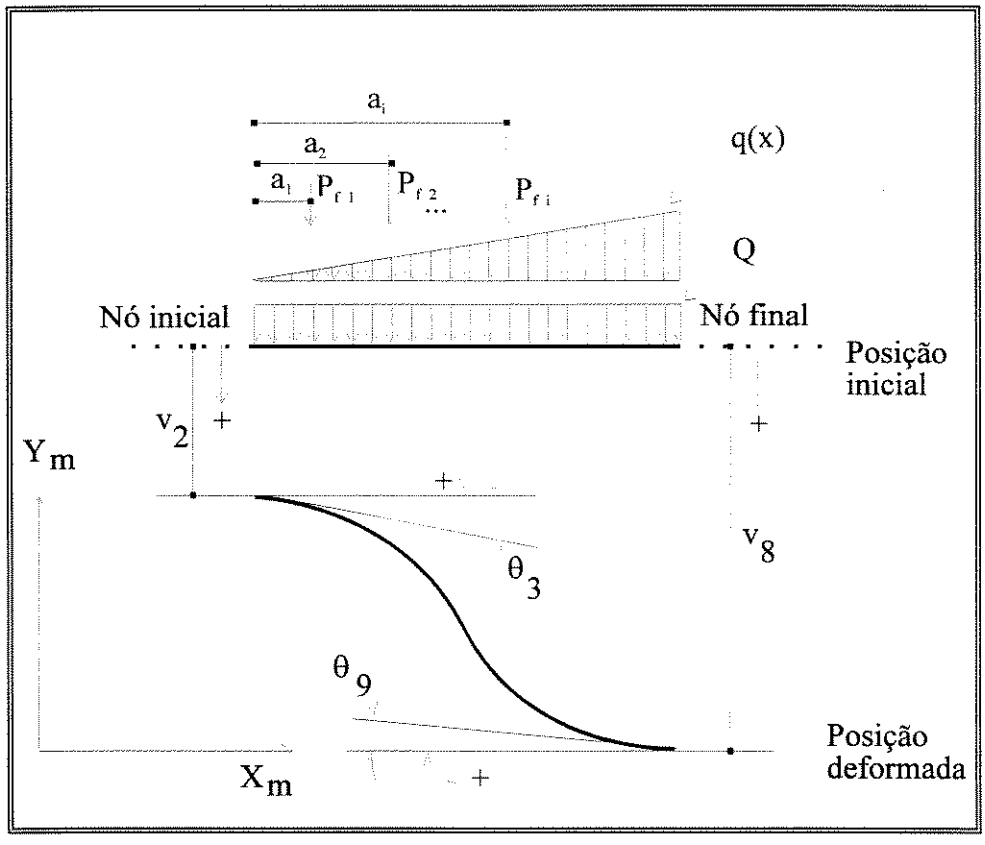

Figura 4.4 - Trecho de viga no plano **XY** com deslocamentos $\mathbf{v_2}$, θ_3 ,$\mathbf{v_8}$ e θ_9 nas extremidades e sujeita as cargas concentradas $\mathbf{P_{f1}}$...$\mathbf{P_{fi}}$, uniformemente distribuida **Q** e linearmente distribuida **q(x)**

Para se estabelecer as funções de interpolação adota-se uma função admissível na forma

$$\bar{v}(x) = a\,x^3 + b\,x^2 + c\,x + d \qquad (4.12)$$

ou, transformando-se para coordenada adimensional como descreve VIZOTTO[1994], onde

$$\xi = \frac{x}{L},$$

$$x = L\,\xi$$

e

$$d\,x = L\;d\,\xi,$$

tem-se

$$\bar{v}(\xi) = a\;L^3\,\xi^3 + b\;L^2\,\xi^2 + c\;L\,\xi + d \qquad (4.13).$$

Impondo-se as 4 condições de contorno do elemento de viga à equação 4.13 , resulta:

1ª condição de contorno $\Rightarrow$ translação no extremo inicial do trecho da viga é v_2, assim

$$\bar{v}(0) = v_2 \quad \Rightarrow \quad d = v_2$$

2ª condição de contorno $\Rightarrow$ rotação no extremo inicial do trecho da viga é θ_3 , assim

$$\bar{\theta}(0) = \frac{d}{d\,x}\left(\bar{v}(0)\right) = \frac{d}{d\xi}\left(\bar{v}(0)\right)\frac{d\xi}{d\,x} = \frac{1}{L}\left(3aL^3\xi^{\;2} + 2bL^2\xi + cL\right)\Big|_{\xi=0} = \theta_{\;3} \quad \Rightarrow \quad c = \theta_3$$

3ª condição de contorno $\Rightarrow$ translação no extremo final do trecho da viga é v_8, assim

$$\bar{v}(1) = v_8 \;\Rightarrow\; a\,L^3\,1^3 + b\,L^2 1^2 + c\,L\,1 + d = v_8 \;\Rightarrow\; a\,L^3 + b\,L^2 = v_8 - v_2 - \theta_3\,L \quad (I)$$

4ª condição de contorno $\Rightarrow$ rotação no extremo final do trecho da viga é θ_9, assim

$$\bar{\theta}(1) = \frac{d}{dx}\left(\bar{v}(1)\right) = \frac{d}{d\xi}\left(\bar{v}(1)\right)\frac{d\xi}{dx} = \frac{1}{L}\left(3\,aL^3\xi^2 + 2bL^2\xi + cL\right)\Big|_{\xi=1} = \theta_9 \Rightarrow 3aL^2 + 2bL = \theta_9 - \theta_3 \quad (II)$$

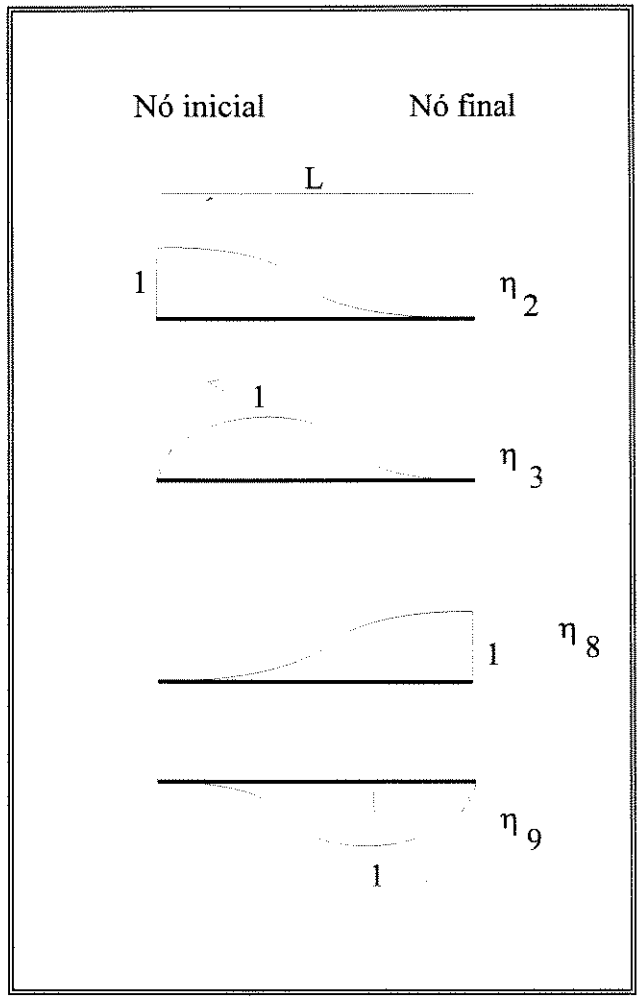

Figura 4.5 - Funções de Interpolação η_2, η_3, η_8 e η_9

Resolvendo-se o sistema formado pelas equações (I) e (II) encontra-se:

$$a = \frac{2}{L^3}(v_2 - v_8) + \frac{\theta_3 + \theta_9}{L^2}$$

e

$$b = \left(\frac{-2\theta_3 - \theta_9}{L}\right) - \frac{3}{L^2}(v_2 - v_8)$$

Substituindo-se os valores encontrados para **a**, **b**, **c** e **d** na equação 4.13, resulta

$$\bar{v}(\xi) = \left[\frac{2}{L^3}(v_2 - v_8) + \frac{\theta_3 + \theta_9}{L^2}\right] L^3 \xi^3 + \left[\left(\frac{-2\theta_3 - \theta_9}{L}\right) - \frac{3}{L^2}(v_2 - v_8)\right] L^2 \xi^2 + \theta_3 \; L\,\xi + v_2$$

Desenvolvendo-se e reagrupando em função dos deslocamentos incógnitos, tem-se a seguinte expressão para a função aproximadora da equação 4.11, porém agora escrita com o parâmetro adimensional ξ:

$$\bar{v}(\xi) = \left(2\xi^3 - 3\xi^2 + 1\right) v_2 + \left(\xi^3 - 2\xi^2 + \xi\right) L\,\theta_3 + \left(-2\xi^3 + 3\xi^2\right) v_8 + \left(\xi^3 - \xi^2\right) L\,\theta_9 \qquad \text{(eq. 4.14)}$$

Comparando-se a equação 4.14 com a equação 4.11 encontra-se

$$\left.\begin{aligned} \eta_2(\xi) &= 2\xi^3 - 3\xi^2 + 1 \\ \eta_3(\xi) &= \left(\xi^3 - 2\xi^2 + \xi\right).L \\ \eta_8(\xi) &= -2\xi^3 + 3\xi^2 \\ \eta_9(\xi) &= \left(\xi^3 - \xi^2\right).L \end{aligned}\right\} \qquad (4.15)$$

de onde são obtidas as primeiras e segundas derivadas:

$$\left.\begin{aligned} \frac{\partial \eta_2}{\partial \xi} &= 6\xi^2 - 6\xi & \frac{\partial^2 \eta_2}{\partial \xi^2} &= 12\xi - 6 \\ \frac{\partial \eta_3}{\partial \xi} &= \left(3\xi^2 - 4\xi + 1\right).L & \frac{\partial^2 \eta_3}{\partial \xi^2} &= \left(6\xi - 4\right).L \\ \frac{\partial \eta_8}{\partial \xi} &= -6\xi^2 + 6\xi & \frac{\partial^2 \eta_8}{\partial \xi^2} &= -12\xi + 6 \\ \frac{\partial \eta_9}{\partial \xi} &= \left(3\xi^2 - 2\xi\right).L & \frac{\partial^2 \eta_9}{\partial \xi^2} &= \left(6\xi - 2\right).L \end{aligned}\right\} \quad (4.16)$$

Assim, tem-se as seguintes relações:

$$\left.\begin{aligned} \frac{d\left(\bar{v}(\xi)\right)}{d\xi} &= \left(6\xi^2 - 6\xi\right) v_2 + \left(3\xi^2 - 4\xi + 1\right).L\theta_3 + \left(-6\xi^2 + 6\xi\right) v_8 + \left(3\xi^2 - 2\xi\right).L\,\theta_9 \\ \frac{d^2\left(\bar{v}(\xi)\right)}{d\xi^2} &= \left(12\xi - 6\right) v_2 + \left(6\xi - 4\right).L\,\theta_3 + \left(-12\xi + 6\right) v_8 + \left(6\xi - 2\right).L\,\theta_9 \end{aligned}\right\} \text{(eq. 4.17)}$$

4.1.2.b) Expressão da Energia de Deformação

Considere-se a expressão da energia de deformação dada pela equação 4.4 aqui reescrita

$$U = \int_V \frac{1}{2} \sigma \varepsilon \, dV \qquad \text{(eq. 4.4)}$$

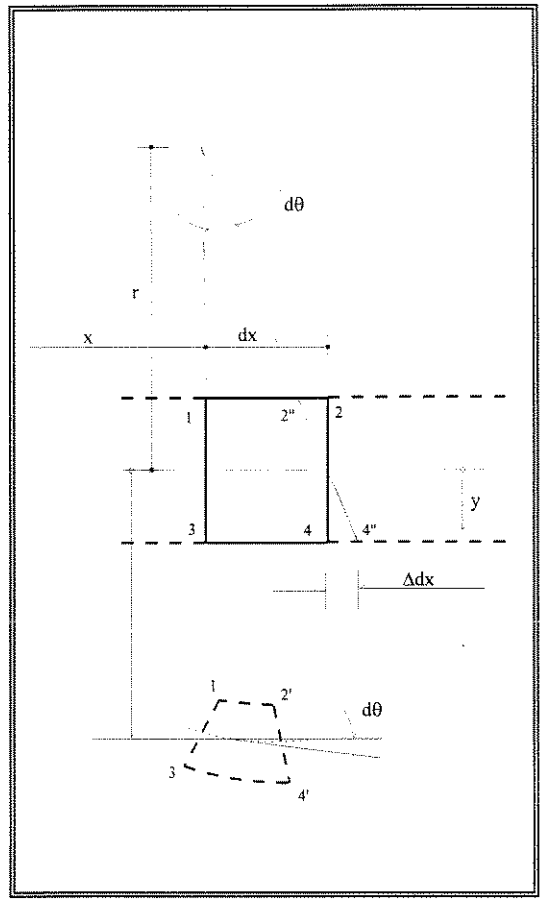

Figura 4.6 - Deformações em um trecho *dx* de um elemento de viga sobre flexão

Na figura 4.6 percebe-se, por semelhança de triângulos, a seguinte relação:

$$\frac{\text{sen } d\theta}{\cos d\theta} = \frac{\Delta \, dx}{y}$$

Considerando-se que para pequenos deslocamentos pode ser feita a seguinte aproximação

$$\text{sen } d\theta \cong d\theta \quad \text{e} \quad \cos d\theta \cong 1$$

tem-se que

$$d\theta = \frac{\Delta\, dx}{y}$$

de onde resulta

$$\Delta\, dx = y\, d\theta$$

Com base na expressão acima e lembrando-se que

$$\varepsilon = \frac{\Delta\, dx}{dx},$$

tem-se

$$\varepsilon\, dx = y\, d\theta$$

ou seja,

$$\varepsilon = y\, \frac{d\theta}{dx}$$

Sendo, também, a seguinte relação

$$\theta(x) = \frac{d}{dx}\left(\bar{v}(x)\right)$$

conclui-se que

$$\varepsilon\,(x) = y \frac{d^2}{dx^2}\left(\bar{v}\,(x)\right) \qquad \text{(eq. 4.18)}$$

Sabe-se, também, que para variação linear entre a tensão e a deformação vale a relação

$$\sigma = E\, \varepsilon\,,$$

portanto, substituindo 4.18 nesta relação resulta

$$\sigma\,(x) = E\, y \frac{d^2}{dx^2}\left(\bar{v}\,(x)\right) \qquad \text{(eq. 4.19)}$$

Substituindo-se na equação 4.4 os expressões da tensão e deformação obtidos respectivamente pelas equações 4.19 e 4.18, tem-se

$$U_{Flexão\,XY} = \int_0^L \left(\int_A \left[\frac{1}{2} E\, y \frac{d^2}{dx^2}\left(\bar{v}\,(x) \right) y \frac{d^2}{dx^2}\left(\bar{v}\,(x) \right) \right] dA \right) dx \qquad \text{(eq. 4.20)}$$

Reescrevendo 4.20, resulta

$$U_{Flexão\,XY} = \int_0^L \left\{ \frac{1}{2} E \left[\frac{d^2}{dx^2}\left(\bar{v}\,(x) \right) \right]^2 \left(\int_A y^2\, dA \right) \right\} dx \qquad \text{(eq. 4.21)}$$

Chamando-se de $\mathbf{J_z}$ o termo

$$\int_A y^2\, dA \Big)$$

da equação 4.21 (denominado momento de inércia em relação ao eixo **z**), tem-se

$$U_{Flexão\,XY} = \int_0^L \left\{ \frac{1}{2} E\, J_z \left[\frac{d^2}{dx^2}\left(\bar{v}\,(x) \right) \right]^2 \right\} dx \qquad \text{(eq. 4.22)}$$

Assim, considerando-se a seção transversal constante ao longo do comprimento a equação 4.22 torna-se

$$U_{Flexão\,XY} = \frac{1}{2} E\, J_z \int_0^L \left[\frac{d^2}{dx^2}\left(\bar{v}\,(x) \right) \right]^2 dx \qquad \text{(eq. 4.23)}$$

A expressão 4.23 fornece o valor da energia de deformação para a função aproximadora adotada em função da coordenada x. Porém, como $\bar{v}$ está escrita com o parâmetro adimensional ξ é necessária uma mudança no integrando bem como nos limites da integração. Assim procedendo, encontra-se

$$U_{Flexão\,XY} = \frac{1}{2} E\, J_z \int_0^L \left[\frac{d}{dx}\left(\frac{d\,\bar{v}\,(\xi)}{d\xi} \right) \frac{d\xi}{dx} \right]^2 dx = \frac{1}{2} E\, J_z \int_0^1 \left[\frac{d^2\left(\bar{v}\,(\xi) \right)}{d\xi^2} \left(\frac{d\xi}{dx} \right)^2 \right]^2 L\, d\xi \quad \text{(eq. 4.24)}$$

Lembrando que

$$\left(\frac{d\xi}{dx}\right)^2 = \frac{1}{L^2},$$

resulta

$$U_{\text{Flexão XY}} = \frac{E J_z}{2 L^3} \int_0^1 \left[\frac{d^2\left(\bar{v}(\xi)\right)}{d\xi^2} \right]^2 d\xi \qquad \text{(eq. 4.25)}$$

Substituindo-se na equação 4.25 a expressão da segunda derivada da função $\bar{v}(\xi)$ apresentada nas relações 4.17 tem-se

$$U_{\text{Flexão XY}} = \frac{E J_z}{2 L^3} \int_0^1 \left[(12\xi - 6)\, v_2 + (6\xi - 4).L\, \theta_3 + (-12\xi + 6)\, v_8 + (6\xi - 2).L\, \theta_9 \right]^2 d\xi \quad \text{(eq. 4.26)}$$

Desenvolvendo-se o quadrado do polinômio e procedendo-se a integração da equação 4.26, resulta finalmente

$$U_{\text{Flexão XY}} = \frac{E J_z}{L^3}\Big(6v_2^2 + 6Lv_2\theta_3 - 12v_2v_8 + 6Lv_2\theta_9 + 2L^2\theta_3^2 - 6Lv_8\theta_3 + \\ + 2L^2\theta_3\theta_9 + 6v_8^2 - 6Lv_8\theta_9 + 2L^2\theta_9^2 \Big) \qquad \text{(eq. 4.27)}$$

A expressão 4.27 fornece a energia de deformação da viga sujeita a flexão em função do vetor de deslocamentos incógnitos

$$\left\{ \begin{matrix} - \\ v \end{matrix} \right\}^T = \{ v_2, \theta_3, v_8, \theta_9 \}.$$

4.1.2.c) Energia Potencial do Trabalho das Forças Externas

Considerando-se a aplicação de três carregamentos diferentes: forças concentradas $\mathbf{P_{f1}}...\mathbf{P_{fi}}$ aplicadas a uma distância $\mathbf{a_1}...\mathbf{a_i}$ da origem, carga **Q** distribuída uniformemente ao longo de todo o comprimento e carga distribuída **q(x)** variando linearmente ao longo de todo o comprimento; e sendo a expressão do trabalho das forças externas a força vezes o deslocamento, analisa-se em separado cada carga aplicada para depois fazer-se a contribuição de cada uma delas em uma única expressão que forneça o trabalho dessas forças externas. Assim, sendo o carregamento apresentado na figura 4.4, tem-se que o trabalho total das forças externas será dado por

$$\Omega_{\text{Flexão XY}} = \Omega_{P_{fi}} + \Omega_Q + \Omega_{q(x)} \qquad \text{(eq. 4.28)}$$

onde

$\Omega_{P_{fi}}$ = trabalho das forças concentradas
Ω_Q = trabalho da força uniformemente distribuida
$\Omega_{q(x)}$ = trabalho da força linearmente distribuida

4.1.2.c.1-) Trabalho realizado pelas forças concentradas P_{fi}

O trabalho realizado pelas forças concentradas $\mathbf{P_{fi}}$ é dado por

$$\Omega_{P_{fi}} = -\sum_{j=1}^{i} P_{fj}\ \bar{v}\left(a_j\right)\ ^{4}$$

Assim, substituindo-se nessa expressão a equação da função aproximadora (equação 4.14), tem-se

$$\Omega_{P_{fi}}(\xi) = -\sum_{j=1}^{i} P_{fj}\left[\left(2\xi_j^{\,3} - 3\xi_j^{\,2} + 1\right) v_2 + \left(\xi_j^{\,3} - 2\xi_j^{\,2} + \xi_j\right) \mathrm{L}\ \theta_3 + \right.$$
$$\left. + \left(-2\xi_j^{\,3} + 3\xi_j^{\,2}\right) v_8 + \left(\xi_j^{\,3} - \xi_j^{\,2}\right) \mathrm{L}\ \theta_9\right]$$
(eq. 4.29)

[4] A ocorrência do sinal negativo nesta e nas demais expressões para o trabalho realizado pelas forças externas está explicada no capítulo 3.

O ponto de aplicação da carga é em $\mathbf{x_j} = \mathbf{a_j}$. Lembrando que

$$x_j = L\,\xi_j ,$$

tem-se que

$$\xi_j = \frac{x_j}{L} = \frac{a_j}{L} .$$

Substituindo na equação 4.29 resulta

$$\Omega_{P_{fi}}(\xi) = -\sum_{j=1}^{i} P_{fj}\left[\left(2\frac{a_j^3}{L^3} - 3\frac{a_j^2}{L^2} + 1\right)v_2 + \left(\frac{a_j^3}{L^3} - 2\frac{a_j^2}{L^2} + \frac{a_j}{L}\right)L\,\theta_3 + \left(-2\frac{a_j^3}{L^3} + 3\frac{a_j^2}{L^2}\right)v_8 + \left(\frac{a_j^3}{L^3} - \frac{a_j^2}{L^2}\right)L\,\theta_9\right] \qquad \text{(eq. 4.30)}$$

Cancelando-se alguns numeradores e denominadores em 4.30 tem-se

$$\Omega_{P_{fi}} = -\sum_{j=1}^{i} P_{fj}\left[\left(2\frac{a_j^3}{L^3} - 3\frac{a_j^2}{L^2} + 1\right)v_2 + \left(\frac{a_j^3}{L^2} - 2\frac{a_j^2}{L} + a_j\right)\theta_3 + \left(-2\frac{a_j^3}{L^3} + 3\frac{a_j^2}{L^2}\right)v_8 + \left(\frac{a_j^3}{L^2} - \frac{a_j^2}{L}\right)\theta_9\right] \qquad \text{(eq. 4.31)}$$

4.1.2.c.2-) Trabalho realizado pela carga uniformemente distribuída Q

O trabalho realizado pela força uniformenente distribuída **Q** será dado por

$$\Omega_Q = -\int_0^L (Q\ \bar{v}(x))\,dx$$

Alterando-se o integrando bem como o limite de integração, tem-se

$$\Omega_Q = -\int_0^1 (Q\ \bar{v}(\xi))\,L\,d\xi \qquad \text{(eq. 4.32)}$$

Assim, substituindo-se nessa expressão a equação da função aproximadora (equação 4.14) resulta

$$\Omega_Q = -\int_0^1 [\ Q\left(\left(2\xi^3 - 3\xi^2 + 1\right) v_2 + \left(\xi^3 - 2\xi^2 + \xi\right) L\,\theta_3 + \right.$$
$$\left. + \left(-2\xi^3 + 3\xi^2\right) v_8 + \left(\xi^3 - \xi^2\right) L\,\theta_9 \right)]\ L\ d\xi \quad \text{(eq. 4.33)}$$

Integrando-se a equação 4.33 tem-se a equação procurada para o trabalho realizado por uma carga uniformente distribuída ao longo de uma viga em função dos deslocamentos incógnitos apresentada abaixo

$$\Omega_Q = -\,Q\,L\left(\frac{1}{2}v_2 + \frac{1}{12}L\theta_3 + \frac{1}{2}v_8 - \frac{1}{12}L\theta_9\right) \qquad \text{(eq. 4.34)}$$

4.1.2.c.3-) Trabalho realizado pela carga distribuída linearmente q(x)

O trabalho realizado pela força linearmenente distribuída **q(x)** será dado por

$$\Omega_{q(x)} = -\int_0^L (\,q(x)\ \bar{v}(x))\,dx \qquad \text{(eq. 4.35)}$$

Sendo variação linerar de carregamento iniciando em zero e terminando em um valor máximo **q**, escreve-se **q(x)** como

$$q(x) = q\frac{x}{L}$$

Colocando-se o carregamento em função do parâmetro admensional ξ, tem-se

$$q(\xi) = q$$

Alterando-se o integrando bem como o limite de integração da equação 4.35 e colocando-se a função aproximadora, resulta

$$\Omega_{q(\xi)} = -\int_0^1 (q\,\xi\,\bar{v}(\xi))\,L\,d\xi \qquad \text{(eq. 4.36)}$$

Assim, substituindo-se nessa expressão a equação da função aproximadora (equação 4.14) tem-se

$$\Omega_{q(\xi)} = -\int_0^1 [\; q\,\xi \left(\left(2\xi^3 - 3\xi^2 + 1\right) v_2 + \left(\xi^3 - 2\xi^2 + \xi\right) L\,\theta_3 + \right.$$
$$\left. + \left(-2\xi^3 + 3\xi^2\right) v_8 + \left(\xi^3 - \xi^2\right) L\,\theta_9 \right)]\; L\; d\xi \quad \text{(eq. 4.37)}$$

Integrando-se a equação 4.37, resulta

$$\Omega_q = -\frac{3qL}{20} v_2 - \frac{qL^2}{30}\theta_3 - \frac{7qL}{20} v_8 + \frac{qL^2}{20}\theta_9 \qquad \text{(eq. 4.38)}$$

Agora, a expressão da energia potencial total devido à flexão no plano XY será a soma das expressões da energia de deformação obtida pela expressão 4.27 com uma equação que forneça o trabalho das três forças externas consideradas, ou seja, a soma das expressões 4.31, 4.34 e 4.38.

Para a consideração da flexão no plano XZ, pode-se tratá-la de maneira semelhante ao procedimento utilizado para a flexão anteriormente estudada. Utiliza-se, neste caso, os mesmo tipo de carregamento e as mesmas equações desenvolvidas, apenas alterando-se o vetor de deslocamentos para que fique compatível com a figura 4.7.

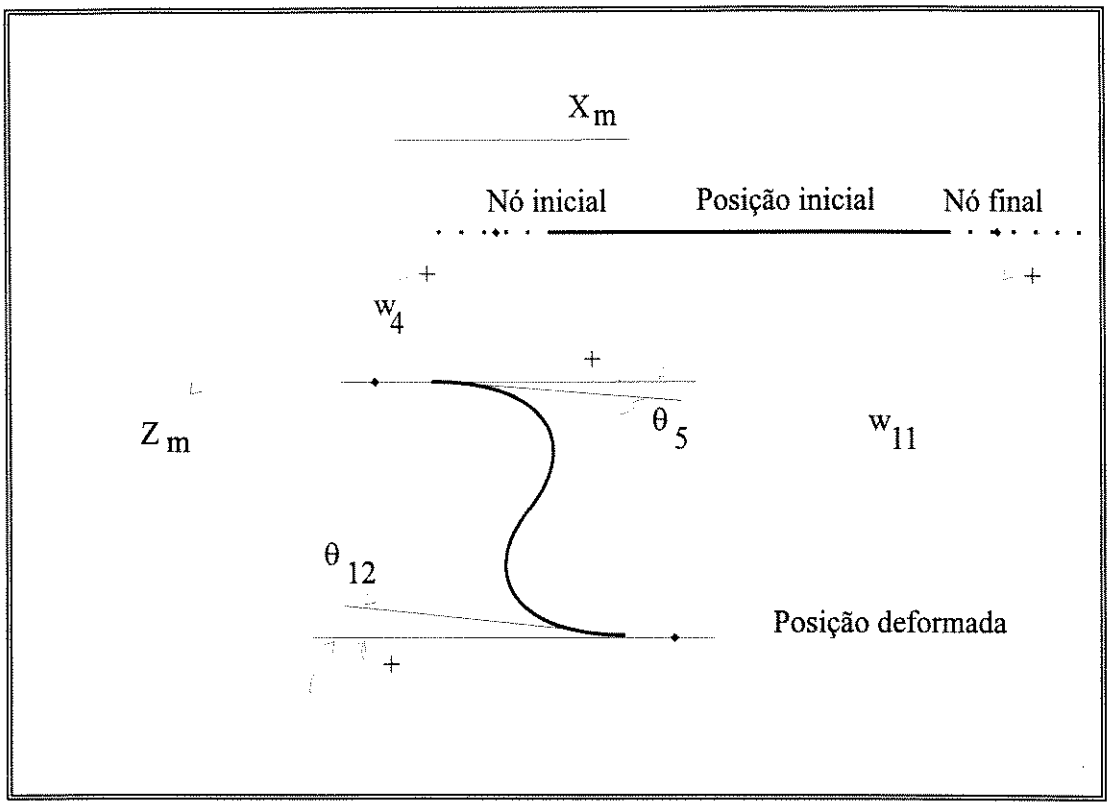

Figura 4.7 - Trecho de uma viga com deslocamentos no plano **XZ** w_4 , θ_5 ,w_{10} e θ_{11} nas extremidades

Assim, o vetor de deslocamentos será

$$\left\{\begin{matrix}-\\v\end{matrix}\right\}^T = \{w_4, \theta_5, w_{10}, \theta_{11}\}$$

bem como a expressão para a energia de deformação (equação 4.27) e as expressões do trabalho realizado pelas forças consideradas (equações 4.31, 4.34 e 4.38) permanecem semelhantes, alterando-se apenas as incognitas nodais (os deslocamentos nas extremidades) e o parâmetro

$$J_z = \int_A y^2 dA$$

que é substituido por

$$J_y = \int_A z^2 dA$$

(momento de inércia em relação ao eixo Y), ou seja:

Expressão da a energia de deformação para flexão no plano XZ

$$\text{Flexão XZ} = \frac{E J_y}{L^3}\left(6w_4^{\,2} + 6Lw_4\theta_5 - 12w_4w_{10} + 6Lw_4\theta_{11} + 2L^2\theta_5^{\,2} - 6Lw_{10}\theta_5 + \right.$$
$$\left. + 2L^2\theta_5\theta_{11} + 6w_{10}^{\,2} - 6Lw_{10}\theta_{11} + 2L^2\theta_{11}^{\,2}\right) \qquad \text{(eq. 4.39)}$$

Expressão do trabalho das forças concentradas (chamando de $\boldsymbol{F_{fi}}$ as cargas concentradas e de $\boldsymbol{d_i}$ as distâncias dos respectivos pontos de aplicação à origem do eixo)

$$\Omega_{F_{fi}} = -\sum_{j=1}^{i} F_{fj}\left[\left(2\frac{d_j^{\,3}}{L^3} - 3\frac{d_j^{\,2}}{L^2} + 1\right)w_4 + \left(\frac{d_j^{\,3}}{L^2} - 2\frac{d_j^{\,2}}{L} + d_j\right)\theta_5 + \right.$$
$$\left. + \left(-2\frac{d_j^{\,3}}{L^3} + 3\frac{d_j^{\,2}}{L^2}\right)w_{10} + \left(\frac{d_j^{\,3}}{L^2} - \frac{d_j^{\,2}}{L}\right)\theta_{11}\right]$$
(eq. 4.40)

*Expressão do trabalho da força uniformemente distribuida **S***

$$\Omega_S = -\,S\,L\left(\frac{1}{2}w_4 + \frac{1}{12}L\theta_5 + \frac{1}{2}w_{10} - \frac{1}{12}L\theta_{11}\right) \qquad \text{(eq. 4.41)}$$

*Expressão do trabalho da força linearmente distribuida (variando de zero a **r** no trecho)*

$$\Omega_{r(x)} = -\frac{3rL}{20}w_4 - \frac{rL^2}{30}\theta_5 - \frac{7rL}{20}w_{10} + \frac{rL^2}{20}\theta_{11} \qquad \text{(eq. 4.42)}$$

Como no caso da flexão no plano **XY**, a energia potencial total será a soma das parcelas dadas pelas equações 4.39, 4.40, 4.41 e 4.42.

4.1.3 - PARCELA DEVIDO AO ESFORÇO TORÇOR E DESLOCAMENTOS CORRESPONDENTES

Para a consideração da contribuição da torção na expressão do funcional da energia potencial total de um elemento de barra, considera-se que a seção não sofre distorção em seu plano (é rígida nele). Assim sendo, seja um trecho genérico de uma viga, apresentado na figura 4.8, com condições de contorno (deslocamentos dos extremos) formando o vetor

$$\left\{ \begin{matrix} - \\ v \end{matrix} \right\}^T = \left\{ \varphi_6 \, , \, \varphi_{12} \right\}$$

e sujeita a atuação de um momento de torção $\mathbf{M_t}$ atuando a uma distância **b** da origem dos eixos.

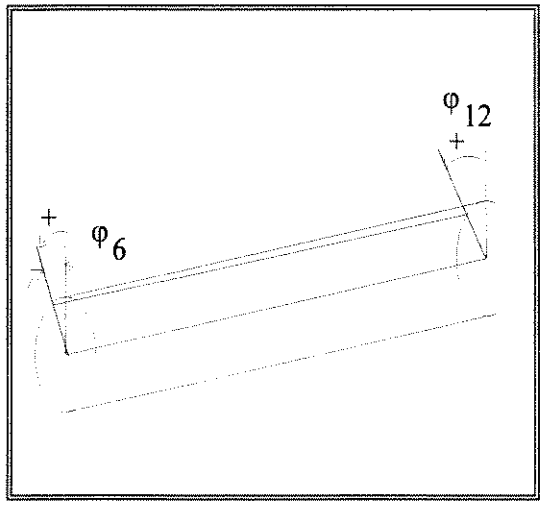

Figura 4.8 - Trecho de Uma Viga Com Deslocamentos φ_6 e φ_{12} nas Extremidades no Plano YZ

4.1.3.a) Função aproximadora

Para a obtenção da solução aproximada da variação do ângulo de rotação, considera-se uma variação linear para este ao longo do comprimento da barra. Tem-se, então, que o espaço vetorial das funções de deslocamentos possíveis é substituído pelo sub-espaço vetorial dos polinômios de 1º grau $\bar{\varphi}(x)$, tal que

$$\bar{\varphi}(x) = \varphi_6\, \eta_6(x) + \varphi_{12}\, \eta_{12}(x) \qquad \text{(eq. 4.43)}$$

onde $\eta_1(x)$ e $\eta_2(x)$ são variações lineares definidas com as seguintes condições de contorno:

$$\bar{\varphi}(0) = \varphi_6 \Rightarrow \begin{cases} \eta_6(0) = 1 \\ \eta_{12}(0) = 0 \end{cases} \quad \text{e} \qquad \bar{\varphi}(L) = \varphi_{12} \Rightarrow \begin{cases} \eta_6(L) = 0 \\ \eta_{12}(L) = 1 \end{cases}$$

Assim, com uma regra de três semelhante à executada para a função aproximadora da deformação devido ao deslocamento normal (figura 4.3), a função aproximadora da variação do ângulo de torção (eq. 4.1) assume a forma

$$\bar{\varphi}(x) = \varphi_6 \left(1 - \frac{x}{L}\right) + \varphi_{12} \left(\frac{x}{L}\right) \qquad \text{(eq. 4.44)}$$

Nota-se que a derivada da expressão 4.44 em relação à x resulta em

$$\frac{d}{dx}\left(\bar{\varphi}(x)\right) = \frac{\varphi_{12} - \varphi_6}{L} \qquad \text{(eq. 4.45)}$$

que representa a variação do ângulo por unidade de comprimento.

4.1.3.b) Expressão da Energia de Deformação

Considere-se a expressão da energia de deformação dada pela equação 4.4 aqui reescrita

$$U = \int_V \frac{1}{2} \gamma \ \tau \, dV \qquad \text{(eq. 4.46)}$$

Para uma variação linear da tensão com a deformação tem-se

$$\gamma = \frac{\tau}{G} \qquad \text{(eq. 4.47)}$$

onde **G** é chamado módulo de elasticidade transversal e guarda a seguinte relação

$$G = \frac{1}{2} \frac{E}{(1+\nu)}$$

sendo E = Módulo de Elasticidade e

ν = Coeficiente de Poisson

A expressão para determinação da tensão cisalhante devido a aplicação de um momento de torção $\mathbf{M_t}$ é

$$\tau = \frac{2M_t}{J_t} c_r \qquad \text{(eq. 4.48)}$$

onde $\mathbf{J_t}$ é chamado constante de torção.

A equação 4.48 fornece o valor da tensão em função de uma ordenada $\mathbf{c_r}$ que é perpendicular ao esqueleto da seção e tem origem nele, como mostra a figura 4.9.

Substituindo-se os valores de γ e τ obtidos nas expressões 4.47 e 4.48, respectivamente, na expressão 4.46, resulta

$$U_{Torção} = \frac{1}{2} \int_V \frac{\tau^2}{G} dV = \frac{1}{2} \int_V \frac{4M_t^2}{G J_t^2} c_r^2 \, dV = 2 \int_V \frac{M_t^2}{G J_t^2} c_r^2 \, dV$$

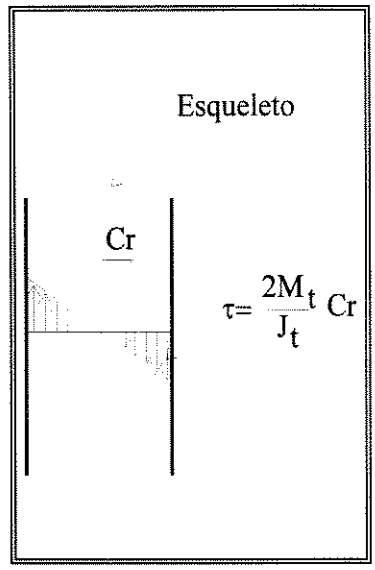

Figura 4.9 - Distribuição de tensões τ ao longo da espessura e orientação da coordenada **cr**

ou seja

$$U_{Torção} = 2\int_0^L \left[\frac{M_t^2}{G\,J_t^2} \int_A \left(c_r^2\right) dA \right] dx \quad \text{(eq. 4.49)}$$

Considerando-se a área como sendo o produto do comprimento do esqueleto pela espessura da seção, transforma-se a integral sobre a área apresentada na equação 4.49 em duas outras integrais: uma sobre o esqueleto e outra sobre a espessura. Tem-se, então, a sequinte expressão

$$U_{Torção} = 2\int_0^L \left\{ \frac{M_t^2}{G\,J_t^2} \left[\int_{esq} \left(\int_{esp} \left(c_r^2\right) dc_r \right) dt \right] \right\} dx \quad \text{(eq. 4.50)}$$

Agora, analisando-se apena a expressão

$$\int_{esp}\left(c_r{}^2\right) dc_r \qquad \text{(eq. 4.51)}$$

que aparece na equação 4.50, tem-se que, sendo **t** a espessura da seção transversal e como $\mathbf{c_r}$ é uma ordenada que tem seu ponto de origem no esqueleto da seção, ou seja, em seu ponto médio, a integração na espessura se processa entre os limites $-\frac{t}{2}$ e $\frac{t}{2}$.

Assim, tem-se para a equação 4.51

$$\int_{esp} c_r{}^2\, dc_r = \int_{-\frac{t}{2}}^{\frac{t}{2}} c_r{}^2\, dc_r = \frac{t^3}{24} + \frac{t^3}{24} = \frac{t^3}{12}$$

Substituindo o valor acima encontrado para a expressão 4.51 na equação 4.50, tem-se

$$U_{Torção} = 2\int_0^L \left[\frac{M_t{}^2}{G\, J_t{}^2} \left(\int_{esq} \frac{t^3}{12}\, dt \right) \right] dx \qquad \text{(eq. 4.52)}$$

que reescrita, torna-se

$$U_{Torção} = \int_0^L \left[\frac{M_t{}^2}{2G\, J_t{}^2} \left(\frac{1}{3} \int_{esq} t^3\, dt \right) \right] dx \qquad \text{(eq. 4.53)}$$

mas,

$$J_t = \frac{1}{3} \int_{esq} t^3\, dt$$

assim, a equação 4.53 torna-se

$$U_{Torção} = \frac{1}{2} \int_0^L \frac{M_t{}^2}{G\, J_t}\, dx \qquad \text{(eq. 4.54)}$$

Sendo a expressão para o momento de torção escrita com a função aproximadora

$$M_t = G\,J_t \frac{d}{dx}\left(\bar{\varphi}\,(x)\right),$$

tem-se que, substituindo em 4.54, resulta

$$U_{Torção} = \frac{1}{2}\int_0^L \frac{\left(G\,J_t \dfrac{d\,\bar{\varphi}\,(x)}{dx}\right)^2}{G\,J_t}\,dx = \frac{1}{2}\int_0^L G\,J_t\left(\frac{d\,\bar{\varphi}\,(x)}{dx}\right)^2 dx \qquad \text{(eq. 4.55)}$$

Considerando-se que a seção permaneça constante ao longo do comprimento e substituindo a expressão da derivada da função aproximadora (obtida pela expressão 4.45), na equação 4.55 tem-se

$$U_{Torção} = \frac{GJ_t}{2}\int_0^L \left(\frac{\varphi_{12} - \varphi_6}{L}\right)^2 dx \qquad \text{(eq. 4.56)}$$

Desenvolvendo-se a expressão 4.56 e integrando-a chega-se na equação que fornece a parcela de contribuição da energia de deformação devido a torção na energia potencial total do elemento. Assim procedendo, resulta

$$U_{Torção} = \left(\varphi_6^2 - 2\,\varphi_6\,\varphi_{12} + \varphi_{12}^2\right)\frac{G\,J_t}{2\,L} \qquad \text{(eq. 4.57)}$$

4.1.3.c) Energia Potencial do Trabalho das Forças Externas

Considera-se várias cargas de torção $\mathbf{M_{t1}}...\mathbf{M_{ti}}$ aplicadas a distâncias $\mathbf{b_1}...\mathbf{b_i}$ da origem do sistema de coordenadas locais.

4.1.3.c.1-) Trabalho realizado pelos momentos de torção

O trabalho realizado pelos momentos de torção $\mathbf{M_{t1}}...\mathbf{M_{ti}}$ aplicados a distâncias $\mathbf{b_1}...\mathbf{b_i}$ da origem dos eixos é dado por

$$\Omega_{M_t} = -\sum_{j=1}^{i} M_{tj}\ \bar{\varphi}\left(b_j\right) \qquad \text{(eq. 4.58)}$$

Assim, substituindo-se nessa expressão o valor da função aproximadora nos pontos de aplicação das cargas (valor da equação 4.44 nos pontos $\mathbf{x} = \mathbf{b_i}$), tem-se

$$\Omega_{M_t} = -\sum_{j=1}^{i} M_{tj}\left[\varphi_{6}\left(1-\frac{b_j}{L}\right)+\varphi_{12}\left(\frac{b_j}{L}\right)\right] \qquad \text{(eq. 4.59)}$$

4.2 - DETERMINAÇÃO DA MATRIZ DE RIGIDEZ PARA UM ELEMENTO DE BARRA NO ESPAÇO TRIDIMENSIONAL

A matriz de rigidez é obtida através da aplicação do princípio da mínima energia ao funcional da energia potencial total do elemento. Para a obtenção do funcional da energia potencial total é necessário que se faça a soma de todas expressões para as parcelas das energias de deformações e para as parcelas dos trabalhos realizados pelos carregamentos supostos atuando no elemento, ou seja,

$$\Pi = U_{\text{Normal}} + \Omega_{\text{Normal}} + U_{\text{Flexão XY}} + \Omega_{Pfi} + \Omega_{Q} + \Omega_{q(x)} + U_{\text{Flexão XZ}} + \Omega_{Ffi} + \Omega_{S} + \Omega_{r(x)} + U_{\text{Torção}} + \Omega_{\text{Mt}} \qquad \text{(eq.4.60)}$$

Assim, somando-se as expressões 4.6, 4.10, 4.27, 4.31, 4.34, 4.38, 4.39, 4.40, 4.41, 4.42, 4.57 e 4.59, obtem-se a expressão em função do vetor de deslocamentos incógnitos

$$\left\{\bar{d}\right\}^{T} = \left\{u_1, v_2, \theta_3, w_4, \theta_5, \varphi_6, u_7, v_8, \theta_9, w_{10}, \theta_{11}, \varphi_{12}\right\}$$

mostrada a seguir

$$\Pi = \frac{EA}{2L}\left(u_1^{\,2} - 2u_1u_7 + u_7^{\,2}\right) + \left\{-\left[\sum_{j=1}^{i} P_j\left(\frac{L-c_j}{L}\right)\right] - \frac{NL}{2}\right\}u_1 - \left[\left(\sum_{j=1}^{i} P_j\frac{c_j}{L}\right) + \frac{NL}{2}\right]u_7 +$$

$$\frac{EJ_z}{L^3}\left(6v_2^{\,2} + 6Lv_2\theta_3 - 12v_2v_8 + 6Lv_2\theta_9 + 2L^2\theta_3^{\,2} - 6Lv_8\theta_3 + \right.$$

$$\left. + 2L^2\theta_3\theta_9 + 6v_8^{\,2} - 6Lv_8\theta_9 + 2L^2\theta_9^{\,2}\right) +$$

$$-\sum_{j=1}^{i} P_{fj}\left[\left(2\frac{a_j^{\,3}}{L^3} - 3\frac{a_j^{\,2}}{L^2} + 1\right)v_2 + \left(\frac{a_j^{\,3}}{L^2} - 2\frac{a_j^{\,2}}{L} + a_j\right)\theta_3 + \left(-2\frac{a_j^{\,3}}{L^3} + 3\frac{a_j^{\,2}}{L^2}\right)v_8 + \left(\frac{a_j^{\,3}}{L^2} - \frac{a_j^{\,2}}{L}\right)\theta_9\right] +$$

$$- \mathrm{Q\,L}\left(\frac{1}{2}v_2 + \frac{1}{12}L\theta_3 + \frac{1}{2}v_8 - \frac{1}{12}L\theta_9\right) - \frac{3qL}{20}v_2 - \frac{qL^2}{30}\theta_3 - \frac{7qL}{20}v_8 + \frac{qL^2}{20}\theta_9 +$$

$$+\frac{EJ_y}{L^3}\left(6w_4^{\,2} + 6Lw_4\theta_5 - 12w_4w_{10} + 6Lw_4\theta_{11} + 2L^2\theta_5^{\,2} - 6Lw_{10}\theta_5 + \right.$$

$$\left. + 2L^2\theta_5\theta_{11} + 6w_{10}^{\,2} - 6Lw_{10}\theta_{11} + 2L^2\theta_{11}^{\,2}\right) +$$

$$-\sum_{j=1}^{i} F_{fj}\left[\left(2\frac{d_j^{\,3}}{L^3} - 3\frac{d_j^{\,2}}{L^2} + 1\right)w_4 + \left(\frac{d_j^{\,3}}{L^2} - 2\frac{d_j^{\,2}}{L} + d_j\right)\theta_5 + \left(-2\frac{d_j^{\,3}}{L^3} + 3\frac{d_j^{\,2}}{L^2}\right)w_{10} + \left(\frac{d_j^{\,3}}{L^2} - \frac{d_j^{\,2}}{L}\right)\theta_{11}\right] +$$

$$- \mathrm{S\,L}\left(\frac{1}{2}w_4 + \frac{1}{12}L\theta_5 + \frac{1}{2}w_{10} - \frac{1}{12}L\theta_{11}\right) - \frac{3rL}{20}w_4 - \frac{rL^2}{30}\theta_5 - \frac{7rL}{20}w_{10} + \frac{rL^2}{20}\theta_{11} +$$

$$+\left(\varphi_6^{\,2} - 2\,\varphi_6\,\varphi_{12} + \varphi_{12}^{\,2}\right)\frac{GJ_t}{2L} - \sum_{j=1}^{i} M_{tj}\left[\varphi_6\left(1 - \frac{b_j}{L}\right) + \varphi_{12}\left(\frac{b_j}{L}\right)\right] \qquad \text{(eq. 4.61)}$$

Para obtenção dos valores das incógnitas nos pontos nodais (vetor de deslocamentos $\left\{\bar{d}\right\}^T$) aplica-se o princípio da mínima energia (condição de estacionaridade do funcional) proposto pela equação 3.12, ou seja, deriva-se parcialmente a equação 4.61 com relação a cada um dos parâmetros incógnitos e iguala-se a zero cada uma das equações assim obtidas.

Dessa forma encontra-se o seguinte sistema de equações:

$$\frac{\partial \Pi}{\partial u_1}=\frac{EA}{L}u_1+0v_2+0\theta_3+0w_4+0\theta_5+0\varphi_6-\frac{EA}{L}u_7+0v_8+0\theta_9+0w_{10}+0\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}P_j\left(\frac{L-c_j}{L}\right)\right]-\frac{NL}{2}$$

$$\frac{\partial \Pi}{\partial v_2}=0u_1+\frac{12EJ_z}{L^3}v_2+\frac{6EJ_z}{L^2}\theta_3+0w_4+0\theta_5+0\varphi_6+0u_7-\frac{12EJ_z}{L^3}v_8+\frac{6EJ_z}{L^2}\theta_9+0w_{10}+0\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}P_{fj}\left(\frac{2a_j^{\,3}}{L^3}-\frac{3a_j^{\,2}}{L^2}+1\right)\right]-\frac{QL}{2}-\frac{3qL}{20}$$

$$\frac{\partial \Pi}{\partial \theta_3}=0u_1+\frac{6EJ_z}{L^2}v_2+\frac{4EJ_z}{L}\theta_3+0w_4+0\theta_5+0\varphi_6+0u_7-\frac{6EJ_z}{L^2}v_8+\frac{2EJ_z}{L}\theta_9+0w_{10}+0\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}P_{fj}\left(\frac{a_j^{\,3}}{L^2}-\frac{2a_j^{\,2}}{L}+a_j\right)\right]-\frac{QL^2}{12}-\frac{qL^2}{30}$$

$$\frac{\partial \Pi}{\partial w_4}=0u_1+0v_2+0\theta_3+\frac{12EJ_y}{L^3}w_4+\frac{6EJ_y}{L^2}\theta_5+0\varphi_6+0u_7-0v_8+0\theta_9-\frac{12EJ_y}{L^3}w_{10}+\frac{6EJ_y}{L^2}\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}F_{fj}\left(\frac{2d_j^{\,3}}{L^3}-\frac{3d_j^{\,2}}{L^2}+1\right)\right]-\frac{SL}{2}-\frac{3rL}{20}$$

$$\frac{\partial \Pi}{\partial \theta_5}=0u_1+0v_2+0\theta_3+\frac{6EJ_y}{L^2}w_4+\frac{4EJ_y}{L}\theta_5+0\varphi_6+0u_7+0v_8+0\theta_9-\frac{6EJ_y}{L^2}w_{10}+\frac{2EJ_y}{L}\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}F_{fj}\left(\frac{d_j^{\,3}}{L^2}-\frac{2d_j^{\,2}}{L}+d_j\right)\right]-\frac{SL^2}{12}-\frac{rL^2}{30}$$

$$\frac{\partial \Pi}{\partial \varphi_6}=0\,u_1+0v_2+0\theta_3+0w_4+0\theta_5+\frac{GJ_t}{L}\varphi_6+0u_7+0v_8+0\theta_9+0w_{10}+0\theta_{11}-\frac{GJ_t}{L}\varphi_{12}-\sum_{j=1}^{i}M_{tj}\left(1-\frac{b_j}{L}\right)$$

$$\frac{\partial \Pi}{\partial u_7}=-\frac{EA}{L}u_1+0v_2+0\theta_3+0w_4+0\theta_5+0\varphi_6+\frac{EA}{L}u_7+0v_8+0\theta_9+0w_{10}+0\theta_{11}+0\varphi_{12}-\left[\left(\sum_{j=1}^{i}P_j\frac{c_j}{L}\right)+\frac{NL}{2}\right]$$

$$\frac{\partial \Pi}{\partial v_8}=0u_1-\frac{12EJ_z}{L^3}v_2-\frac{6EJ_z}{L^2}\theta_3+0w_4+0\theta_5+0\varphi_6+0u_7+\frac{12EJ_z}{L^3}v_8-\frac{6EJ_z}{L^2}\theta_9+0w_{10}+0\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}P_{fj}\left(-\frac{2a_j^{\,3}}{L^3}+\frac{3a_j^{\,2}}{L^2}\right)\right]-\frac{QL}{2}-\frac{7qL}{20}$$

$$\frac{\partial \Pi}{\partial \theta_9}=0u_1+\frac{6EJ_z}{L^2}v_2+\frac{2EJ_z}{L}\theta_3+0w_4+0\theta_5+0\varphi_6+0u_7-\frac{6EJ_z}{L^2}v_8+\frac{4EJ_z}{L}\theta_9+0w_{10}+0\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}P_{fj}\left(\frac{a_j^{\,3}}{L^2}-\frac{a_j^{\,2}}{L}\right)\right]+\frac{QL^2}{12}+\frac{qL^2}{20}$$

$$\frac{\partial \Pi}{\partial w_{10}}=0u_1+0v_2+0\theta_3-\frac{12EJ_y}{L^3}w_4-\frac{6EJ_y}{L^2}\theta_5+0\varphi_6+0u_7-0v_8+0\theta_9+\frac{12EJ_y}{L^3}w_{10}-\frac{6EJ_y}{L^2}\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}F_{fj}\left(-\frac{2d_j^{\,3}}{L^3}+\frac{3d_j^{\,2}}{L^2}\right)\right]-\frac{SL}{2}-\frac{7rL}{20}$$

$$\frac{\partial \Pi}{\partial \theta_{11}}=0u_1+0v_2+0\theta_3+\frac{6EJ_y}{L^2}w_4+\frac{2EJ_y}{L}\theta_5+0\varphi_6+0u_7+0v_8+0\theta_9-\frac{6EJ_y}{L^2}w_{10}+\frac{4EJ_y}{L}\theta_{11}+0\varphi_{12}-\left[\sum_{j=1}^{i}F_{fj}\left(\frac{d_j^{\,3}}{L^2}-\frac{d_j^{\,2}}{L}\right)\right]+\frac{SL^2}{12}+\frac{rL^2}{20}$$

$$\frac{\partial \Pi}{\partial \varphi_{12}}=0\,u_1+0v_2+0\theta_3+0w_4+0\theta_5-\frac{GJ_t}{L}\varphi_6+0u_7+0v_8+0\theta_9+0w_{10}+0\theta_{11}+\frac{GJ_t}{L}\varphi_{12}-\sum_{j=1}^{i}M_{tj}\left(\frac{b_j}{L}\right)$$

$$
\begin{bmatrix}
\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{12EJ_z}{L^3} & \frac{6EJ_z}{L^2} & 0 & 0 & 0 & 0 & -\frac{12EJ_z}{L^3} & \frac{6EJ_z}{L^2} & 0 & 0 & 0 \\
0 & \frac{6EJ_z}{L^2} & \frac{4EJ_z}{L} & 0 & 0 & 0 & 0 & -\frac{6EJ_z}{L^2} & & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{12EJ_y}{L^3} & \frac{6EJ_y}{L^2} & 0 & 0 & 0 & 0 & -\frac{12EJ_y}{L^3} & \frac{6EJ_y}{L^2} & 0 \\
0 & 0 & 0 & \frac{6EJ_y}{L^2} & \frac{4EJ_y}{L} & 0 & 0 & 0 & 0 & -\frac{6EJ_y}{L^2} & \frac{2EJ_y}{L} & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{GJ_t}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{GJ_t}{L} \\
-\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 & \frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{12EJ_z}{L^3} & -\frac{6EJ_z}{L^2} & 0 & 0 & 0 & 0 & \frac{12EJ_z}{L^3} & -\frac{6EJ_z}{L^2} & 0 & 0 & 0 \\
0 & \frac{6EJ_z}{L^2} & \frac{2EJ_z}{L} & 0 & 0 & 0 & 0 & -\frac{6EJ_z}{L^2} & \frac{4EJ_z}{L} & 0 & 0 & 0 \\
0 & 0 & 0 & -\frac{12EJ_y}{L^3} & -\frac{6EJ_y}{L^2} & 0 & 0 & 0 & 0 & \frac{12EJ_y}{L^3} & -\frac{6EJ_y}{L^2} & 0 \\
0 & 0 & 0 & \frac{6EJ_y}{L^2} & \frac{2EJ_y}{L} & 0 & 0 & 0 & 0 & -\frac{6EJ_y}{L^2} & \frac{4EJ_y}{L} & 0 \\
0 & 0 & 0 & 0 & 0 & -\frac{GJ_t}{L} & 0 & 0 & 0 & 0 & 0 & \frac{GJ_t}{L}
\end{bmatrix}
\begin{Bmatrix}
u_1 \\ v_2 \\ \theta_3 \\ w_4 \\ w_4 \\ \varphi_6 \\ u_7 \\ v_8 \\ \theta_9 \\ w_{10} \\ \theta_{11} \\ \varphi_{12}
\end{Bmatrix}
=
\begin{Bmatrix}
\left[\sum_{j=1}^{i} P_j \left(\frac{L-c_j}{L}\right)\right] + \frac{NL}{2} \\
\left[\sum_{j=1}^{i} P_{fj} \left(\frac{2a_j^{\,3}}{L^3} - \frac{3a_j^{\,2}}{L^2} + 1\right)\right] + \frac{QL}{2} + \frac{3qL}{20} \\
\left[\sum_{j=1}^{i} P_{fj} \left(\frac{a_j^{\,3}}{L^2} - \frac{2a_j^{\,2}}{L} + a_j\right)\right] + \frac{QL^2}{12} + \frac{qL^2}{30} \\
\left[\sum_{j=1}^{i} F_{fj} \left(\frac{2d_j^{\,3}}{L^3} - \frac{3d_j^{\,2}}{L^2} + 1\right)\right] + \frac{SL}{2} + \frac{3rL}{20} \\
\left[\sum_{j=1}^{i} F_{fj} \left(\frac{d_j^{\,3}}{L^2} - \frac{2d_j^{\,2}}{L} + d_j\right)\right] + \frac{SL^2}{12} + \frac{rL^2}{30} \\
\left[\sum_{j=1}^{i} M_{tj} \left(1 - \frac{b_j}{L}\right)\right] \\
\left[\sum_{j=1}^{i} P_j \frac{c_j}{L}\right] + \frac{NL}{2} \\
\left[\sum_{j=1}^{i} P_{fj} \left(-\frac{2a_j^{\,3}}{L^3} + \frac{3a_j^{\,2}}{L^2}\right)\right] + \frac{QL}{2} + \frac{7qL}{20} \\
\left[\sum_{j=1}^{i} P_{fj} \left(\frac{a_j^{\,3}}{L^2} - \frac{a_j^{\,2}}{L}\right)\right] - \frac{QL^2}{12} - \frac{qL^2}{20} \\
\left[\sum_{j=1}^{i} F_{fj} \left(-\frac{2d_j^{\,3}}{L^3} + \frac{3d_j^{\,2}}{L^2}\right)\right] + \frac{SL}{2} + \frac{7rL}{20} \\
\left[\sum_{j=1}^{i} F_{fj} \left(\frac{d_j^{\,3}}{L^2} - \frac{d_j^{\,2}}{L}\right)\right] - \frac{SL^2}{12} - \frac{rL^2}{20} \\
\left[\sum_{j=1}^{i} M_{tj} \left(\frac{b_j}{L}\right)\right]
\end{Bmatrix}
$$

Colocando-se as relações anteriores em forma matricial a expressão acima.

A matriz que multiplica o vetor de deslocamentos incógnitos no primeiro membro da equação anterior é a matriz de rigidez do elemento de barra no espaço tridimensional, com dois nós e seis graus de liberdade por nó. Chama-se vetor das cargas nodais equivalentes ao vetor que aparece isolado no segundo membro da mesma equação.

5 - AS CLASSES BÁSICAS DO SISTEMA

O objetivo deste projeto é o desenvolvimento de uma biblioteca de classes de objetos relacionados a estruturas aporticadas tridimensionais, concebido para crescer através da introdução de novas teorias e idéias que poderão ser rapidamente implementadas com a utilização de objetos especializados.

Considerando-se que o código que possibilita o estudo do problema em duas dimensões é semelhante àquele que realiza a análise no espaço tridimensional, foi proposto que a primeira versão do ambiente computacional fosse desenvolvida para executar a análise de estruturas planas, facilitando-se desta forma a localização e a correção de futuros erros, principalmente os lógicos, que podem ocorrer em uma implementação. Com essa diretriz foram desenvolvidas as classes para estruturas planas, tendo-se em vista que essas classes seriam alteradas, adequando-as a estruturas tridimensionais, objetivo inicial deste trabalho.

Porém, considerando-se também outros fatores:

1. O código para análise plana foi completamente desenvolvido estando, portanto, pronto e testado;
2. As vantagens do encapsulamento do objeto e facilidade de manutenção e alteração do código que a OOP oferece através do uso da propriedade da herança;
3. A análise de estruturas no plano utiliza uma quantidade de memória menor que a utilizada para um problema em três dimensões;

optou-se por acrescentar os métodos e classes necessários para o estudo tridimensional sem perder a capacidade do ambiente analisar um problema bidimensional. Possibilita-se, assim, o estudo de um maior número de barras na análise plana que o conseguido no caso tridimensional devido a utilização de uma quantidade menor de memória por elemento de barra (matriz de rigidez menor, uma coordenada a menos

por nó da barra, etc) com um mesmo equipamento , ou um melhor aproveitamento deste em qualquer outro tipo de problema no qual uma das coordenadas do nó seja irrelevante (cálculo das características geométricas de uma seção, por exemplo).

Algumas das classes utilizadas no projeto inicial (**TSqMatrix**, **TNode**, **TList**, etc), foram aqui implementadas sem nenhuma ou com as poucas modificações que se fizeram necessárias durante o desenvolvimento deste ambiente computacional (modificar é prática muito comum no transcorrer da implementação de um software e tarefa executada com uma das facilidades que a filosofia da orientação por objetos oferece), enquanto que outras foram criadas para atender as exigências de um campo mais amplo de análise.

5.1 - UMA FUNÇÃO PARA ERROS

Uma função de implementação simples mas muito utilizada em todo o sistema é a função **error**. Não foi implementada como uma classe devido a natureza de o erro não ser um objeto e ter escopo global (uma mensagem para o usuário e a posibilidade de continuar executando a análise ou ter que abortar o processo onde ocorreu uma falha podem acontecer dentro de qualquer função no programa). Além disso, a função aqui implementada não manipula dados ou variáveis de outras partes do sistema, o que permite utilizá-la em qualquer área do projeto sem a preocupação de que possa provocar erros com manipulação de dados.

Listagem 5. 1 - Parte do arquivo **error.h**

```
#define CONTINUAR 0
#define SAIR 1

void error ( const char * s , unsigned int halt=CONTINUAR) ;
```

Deve-se notar as macros definidas **SAIR** e **CONTINUAR** que são utilizadas pela função para avaliar se devido ao erro ocorrido deve ser abortada a execução ou se esta pode prosseguir dando-se apenas um alerta ao usuário. O valor *default* é **CONTINUAR**, significando que se não for passado para a função nenhum valor além da mensagem a execução da análise continuará. Para interromper a execução deve-se passar o valor **SAIR** na chamada à função, logo após a mensagem que informa o erro ocorrido.

Nesta implementação escolheu-se passar a mensagem como parâmetro para a função, podendo-se reescrevê-la de modo que seu código identifique o erro ocorrido através de valores constantes e únicos como sugerido por SWAN[1993]. Dessa forma, quando em uma parte do programa a função **error** for chamada necessite apenas do valor do código para o erro ocorrido, pois a escolha da mensagem a ser exibida será feita dentro do corpo da função. Deve ser salientado que esse procedimento limita as mensagens àquelas existentes no corpo da função, devendo-se manipular código já estável a cada nova mensagem que se deseje implementar, enquanto que para o procedimento escolhido neste trabalho basta apenas passar a nova mensagem como parâmetro na chamada da função.

5.2 - O PROBLEMA ARMAZENADO EM LISTAS

Os dados relativos a análise de um problema necessitam ser armazenados de alguma forma para que o sistema os utilize a medida do necessário. Para essa finalidade existem várias formas : vetores (utilizado por MACKIE[1992]), arquivos, pilhas, árvores (propostas por SCHOLZ[1992]) e listas (encontradas em SWAN[1993] e outros). Optou-se neste trabalho pelo armazenamento dos dados em listas resultando, assim, em lista de nós, lista de elementos, lista de restrições, etc.

Uma lista aqui proposta pretende ser genérica, podendo ser utilizada em qualquer outro projeto até mesmo sem modificações. Para tanto, o dado que ela armazena e manipula através de várias funções é um ponteiro para um objeto do tipo ***void***, o que permite trabalhar com o endereço de objetos de qualquer tipo, necessitando-se apenas que seja feita uma conversão de tipo (procedimento denominado ***cast***) no endereço a ser recuperado da lista.

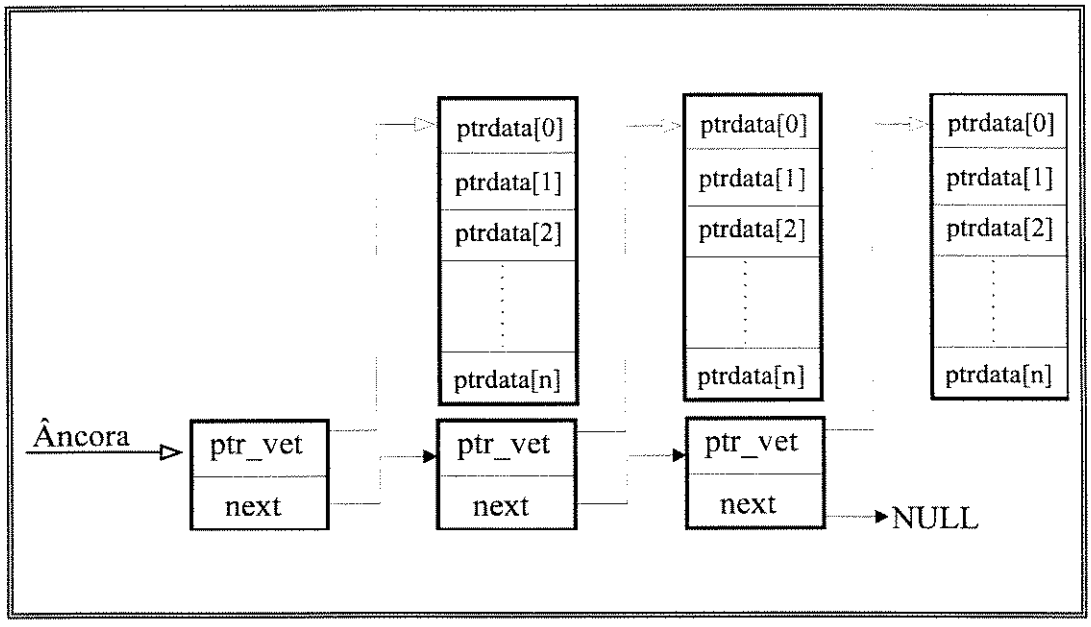

Figura 5.1 - Esquema De Uma Lista de Vetores

Apresenta-se a seguir parte do arquivo de cabeçalho da classe **TList** com uma breve elucidação, logo após a declaração de cada método ou variável, sobre qual a sua função.

Listagem 5. 2 - Parte do arquivo de cabeçalho da classe **TList**

```
# define  ARRAY_SIZE 10
                                        class TList
{
        private:
                unsigned int number_of_array,    // guarda o numero de arrays da lista.
                 size_of_array,                  // guarda o tamanho dos arrays da lista.
                counter, locus;                  // contadores.

                struct Str_Array_List
                   {
                     void * *data ;              // pointer para um array de ponteiros
                                                 // que apontam para objetos de tipo void
                     Str_Array_List *NextArray ;         // pointer para a proxima estrutura de
                                                         // array
                   } ;
                Str_Array_List *CurrentArrayList ;       //ponteiro para a estrutura de array na
                                                 // qual se esta posicionado o contador locus
                Str_Array_List *Ancora ;         //ponteiro para a primeira estrutura de array
                void Goto_End( ) ;                       //posiciona no final da lista
                void Goto_Begin ( ) ;                    //posiciona no inicio da lista
                void Test_And_Aloc_New_List ( ) ;        //verifica se a lista esta cheia. Se
                                                 // estiver, aloca nova estrutura de array

        public:
                TList(unsigned int n = ARRAY_SIZE) ;             // construtora
                ~TList( );       // destrutora == CUIDADO== deve-se deletar todos os itens
                                 // armazenados na lista antes de se proceder a remocao da lista
                void Remove (void *item) ;       // CUIDADO : deve-se deletar o item a ser
                                         // removido da lista antes de proceder a remocao.
                void Insert_Begin ( void *item ) ;       // adiciona um dado no inicio da lista
                void Insert_End ( void *item) ;          // adiciona um dado no final da lista
                void *Get_Next ( ) ;             // recupera o proximo dado armazenado
                void *Get_First ( ) ;            // recupera o primeiro dado armazenado
                unsigned int Get_Used ( ) ;      // retorna o numero de dados armazenados
                unsigned int Get_Capacity ( ) ;  // retorna a capacidade de armazenamento
                                                 // da lista
                void * Get_Data (unsigned int n); // retorna o enesimo dado armazenado na
                                         // lista. O parametro n deve ser fornecido pelo usuario
};
```

Outra proposta no sentido de gerenciar mais eficientemente a quantidade de memória do equipamento é a alocação dinâmica de uma ***struct*** composta por um vetor (***array***) de ponteiros ***void*** e um ponteiro para uma próxima ***struct*** semelhante (Figura 5.1). Dessa forma não existe um ponteiro para o próximo elemento da lista a cada dado armazenado e sim um ponteiro para cada conjunto de dados. A quantidade de dados em cada ***struct*** é definida pelo usuário através de um parâmetro de entrada que deve ser fornecido a cada objeto **TList** que for criado. Analisando-se o código de parte do arquivo de cabeçalho da classe **TList** apresentado, percebe-se que esse parâmetro é definido como ***default*** em 10 unidades de ponteiros ***void*** pela macro **ARRAY_SIZE 10** a qual é passada para a função construtora.

Nota-se pelo parágrafo acima e pelo esquema da figura 5.1 que a lista aqui implementada assemelha-se a um vetor dinâmico, porém, considerando-se que:

- esse vetor dinâmico pode crescer ou encolher à medida das necessidades de armazenamento de dados;
- existe um ponteiro para um próximo bloco de dados (o que inexiste em um vetor);
- o tamanho do bloco de dados pode ser o suficiente para apenas um dado,

sugere um comportamento de lista, neste trabalho convencionou-se tratar um objeto dessa classe como uma lista.

Na implementação da classe **TList** existem métodos declarados na área **private** de forma que somente possam ser utilizados apenas pelas funções da classe. Assim, **Insert_Begin**, **Get_Data** ou **Get_First** por exemplo, fazem uso de **Goto_Begin** ou então, **Insert_End** e **Insert_Begin** fazem uso de **Test_And_Aloc_New_List** (método que testa se a lista está completa e, caso esteja, aloca espaço para um novo vetor com ***n*** ponteiros ***void*** - ***n*** definido pelo usuário ou

default em 10 -), deixando assim todo o trabalho de gerenciamento da lista com a classe.

Especial atenção deve ser observada quando da utilização das funções **~TList** (destrutora) e **Remove**. Devido ser uma lista que guarda objetos genéricos, ao se remover um dado nela armazenado ou quando se deseja eliminar a lista, existe a necessidade de se providenciar a remoção desse dado ou a eliminação de todos eles individualmente, para que seja retornada ao sistema a memória ocupada por esses dados. Para tanto deve-se proceder da seguinte forma:

1. Atribui-se o dado (ponteiro ***void***) a uma variável do tipo do dado armazenado fazendo-se um ***cast***.
2. Se a intenção é remover um dado da lista deve-se, após realizar o procedimento 1, chamar a função **Remove** passando a variável como parâmetro.
3. Chamar a função destrutora para a variável que endereça o dado que se deseja eliminar, retornando ao sistema a quantidade de memória que essa variável ocupava.
4. Se a intenção é eliminar a lista, deve-se executar os procedimentos 1 e 3 para todos os dados que estão armazenados na lista. Só então deve-se chamar a função destrutora da lista.

5.3 - AS CLASSES PARA VETORES

Em diversas partes do ambiente aqui desenvolvido é utilizada a forma de vetores para a armazenagem de dados (vetor de cargas globais, vetor de deslocamentos, etc). Muito embora na linguagem C++ exista uma forma de definir variáveis que representem vetores, projetou-se o desenvolvimento de uma classe cujos objetos tenham o mesmo comportamento de tais vetores deixando, assim, o sistema menos suceptível a erros e possibilitando uma manutenção mais fácil quando esta se fizer necessária.

Implementou-se, então, uma classe mais geral chamada **TVetor** para representar uma matriz coluna (vetor) e dela derivaram-se duas outras: **TVetInt** e **TVetFloat** as quais especificam vetores que armazemam dados do tipo ***int*** (inteiro) e vetores que armazemam dados do tipo ***float*** (real) respectivamente. O esquema de derivação está mostrado na figura 5.2.

Como os métodos das classes não foram declarados como virtuais, um ponteiro para um objeto **TVetor** poderá endereçar qualquer objeto de uma sua classe derivada, porém, caso existam dois métodos com a mesma declaração, aquele que será executado em uma chamada será o da classe raiz.

Listagem 5. 3 - Parte do arquivo de cabeçalho da classe **TVetor**

```
                                class TVetor
{
 protected :
        unsigned int size ;

        public :
                TVetor ( unsigned int order = 0 ) ;        // construtora
                ~TVetor ( ) { } ;                                  // destrutora
                unsigned int Get_Size( ) { return(size) ; }        // retorna a ordem do vetor
} ;
```

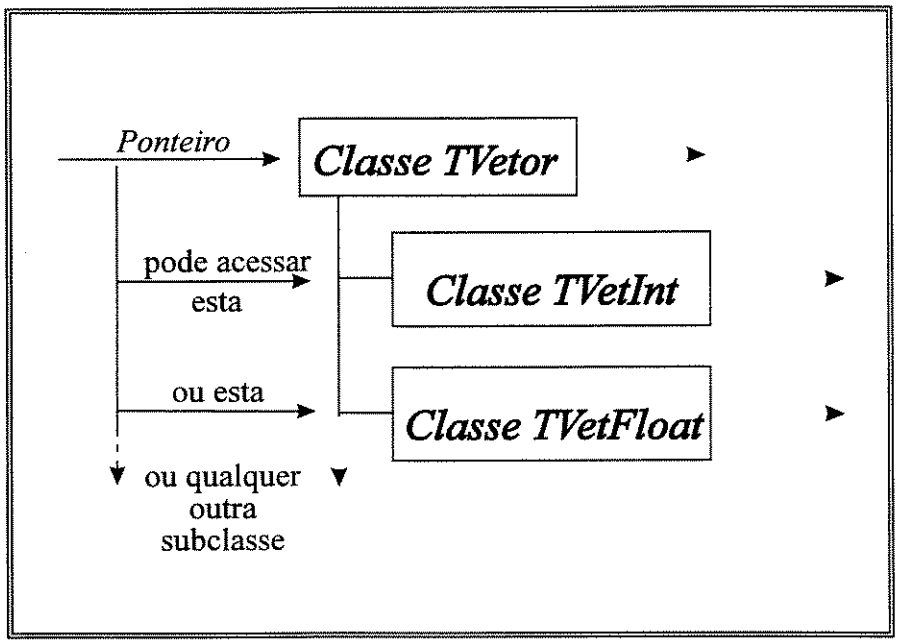

Figura 5.2 - Esquema para as classes que representam vetores

A classe **TVetor** contém na área **protected** um variável denominada **size** a qual determina a quantidade de linhas do vetor (ordem do vetor). O valor dessa variável deve ser fornecida para a função construtora quando da inicialização de um objeto dessa classe ou das suas sub-classes.

Listagem 5. 4 - Parte do arquivo de cabeçalho da classe **TVetInt**

```
                        class TVetInt : public TVetor
{
        protected:
                unsigned int *data ;

        public:
                TVetInt( unsigned int order = 0 ) ;          // construtora
                ~TVetInt( ) ;                                // destrutora
                void Put_Element ( unsigned int value, unsigned int a ) ;    // armazena um valor
                                                             // na posicao a do vetor
                unsigned int Get_Element (unsigned int a ) ;         // recupera o valor
                                                        // armazenado na posicao a do vetor
                void Add_Element ( unsigned int value, unsigned int a ) ;    // adiciona um valor
                                        // ao valor que ja esta armazenado na posicao a do vetor
};
```

Listagem 5. 5 - Parte do arquivo de cabeçalho da classe **TVetFloat**

```
                                class TVetFloat : public TVetor
{
        protected:
                float *data ;

        public:
                TVetFloat( unsigned int order = 0 ) ;                   // construtora
                ~TVetFloat( ) ;                                         // destrutora
                void Put_Element ( float value, unsigned int a ) ;      // armazena um valor na
                                                                        // posicao a do vetor
                float Get_Element (unsigned int a ) ;                   // recupera o valor
                                                        // armazenado na posicao a  do vetor
                void Add_Element ( float value, unsigned int a ) ;      // adiciona um valor ao valor
                                                        // que ja esta arazenado na posicao a do vetor
};
```

As classes **TVetInt** e **TVetFloat**, por serem derivadas de **TVetor**, herdam a variável **size** e o método que retorna a ordem do vetor (método **Get_Size**). Outras funções implementadas para essas classes são:

- **Put_Element** - Armazena um valor em uma posição **i** (linha) do vetor. Deve-se fornecer o valor (**int** para **TVetInt** e **float** para **TVetFloat**) seguido da posição onde ele será armazenado (número da linha).
- **Add_Element** - Adiciona um valor ao valor que já existe armazenado em uma posição **i** (linha) do vetor. Deve-se fornecer o valor a ser adicionado (um **int** para **TVetInt** e um **float** para **TVetFloat**) seguido da posição onde ele será somado e o resultado armazenado.
- **Get_Element** - Retorna o valor armazenado na posição **i** (linha) do vetor. Deve-se fornecer a posição de onde se deseja recuperar o valor armazenado.

5.4 - A CLASSE TSqMatrix

Essa classe, utilizada para representar uma matriz quadrada cheia (sem considerações a respeito de banda ou simetria), contém duas variáveis que estão declaradas na área **protected** : um *unsigned int* denominado **size** que determina o tamanho (ordem) da matriz e uma variável ponteiro para tipo *float* chamada **data**, utilizada para armazenar os dados (valor dos elementos) da matriz de maneira vetorial (*array*) através da fórmula **data[i*size+j]**, onde **i** representa a linha da matriz na qual se encontra o elemento e **j** é a coluna. A função construtora aloca dinamicamente espaço para armazenar os elementos da matriz e preenche esses espaços com zeros, garantindo que qualquer objeto dessa classe está zerado quando de sua inicialização.

Listagem 5. 6 - Parte do arquivo de cabeçalho da classe **TSqMatrix**

```
#define ERRO_MATRIX 0.00001          // usado no metodo de decomposicao para decidir
                                     // se uma matrix e ou nao singular.
                                     class TSqMatrix
{
        protected:
                unsigned int size ;
                float *data ;

        public:
                TSqMatrix(unsigned int order = 0) ;        // construtora
                ~TSqMatrix( ) ;                            // destrutora
                void Put_Element( float value, unsigned int a, unsigned int b);
                float Get_Element(unsigned int a , unsigned int b);
                void Add_Element ( float value, unsigned int a , unsigned int b) ;
                void Solve ( TVetFloat * fvet ) ;
                void Display() ;
                void operator *= (TSqMatrix MatB) ;
                void operator = (TSqMatrix MatB) ;
                unsigned int Get_Size( ) { return(size) ; }    // retorna a ordem da matriz
                void Transposta( ) ;       // calcula a matriz transposta da que fez a chamada ao
                                           // metodo armazenando o resultado nela
};
```

Os métodos **Put_Element** e **Get_Element**, semelhante as funções de mesmo nome das classes **TVetInt** e **TVetFloat,** armazenam e retornam o valor armazenado na posição (**i,j**) da matriz, respectivamente. Para tanto **Put_Element** necessita do valor (**value**) a ser armazenado e dos valore **i** e **j** da posição (**i** é o número da linha e **j** é o número da coluna) onde ocorrerá o armazenamento, enquanto **Get_Element** tem como parâmetros de entrada os indices **i** e **j** da posição na matriz onde se encontra o valor que se deseja recuperar. **Add_Element** é um método semelhante a **Put_Element** tendo os mesmos parâmetros de entrada, porém adicionando o valor passado na chamada da função ao valor que já se encontra armazenado na posição (**i,j**) da matriz.

A função **Solve** necessita de um objeto da classe **TVetFloat** como parâmetro de entrada para poder encontrar a solução do sistema de equações **[A].{b} = {c}**, onde **[A]** é a matriz que chamou o método, **{b}** é o vetor de incógnitas e **{c}** é o vetor que é passado para a função (objeto **TVetFloat**). Na solução desse sistema ocorre a decomposição da matriz que chama a função em uma triangular superior e outra inferior (ambas armazenadas na própria matriz que foi associada ao método). O resultado da solução do sistema (vetor **{b}**) é armazenado no objeto **TVetFloat** que foi passado como parâmetro de entrada para o método (vetor **{c}**).

Esse procedimento acarreta uma perda de tempo de processamento se for necessária uma outra análise da estrutura (caso de uma análise mais detalhada de um dos elementos componentes do modelo - decomposição daquele em 10 por exemplo - ou de um outro vetor de cargas atuantes na estrutura original quando se tem múltiplos carregamentos), isso porque a matriz de rigidez global e os dados do vetor passado como parâmetro de entrada (vetor de cargas globais) foram perdidos e deverão ser lidos de um arquivo onde haviam sido previamente armazenados, a fim de que se possa chamar o método para a solução do novo sistema de equações.

Note-se a macro definida no início do arquivo de cabeçalho **ERRO_MATRIX**. Caso ocorra na decomposição da matriz que um dos elementos da diagonal principal resulte em um valor menor que o definido por essa macro, o método exibirá uma mensagem de erro e encerrará a execução do programa.

Nesta classe encontra-se outra vantagem oferecida pela linguagem C++ : a possibilidade de se sobrescrever as funções dos operadores. Aqui os operadores de igualdade (=) e o de multiplicação com atribuição ao próprio multiplicando (*=) são redefinidos para utilização na atribuição dos elementos de uma matriz à outra que fez a chamada da função (**[A]=[B]** - atribui os elementos de **[B]** à matriz **[A]**) e para utilização na multiplicação entre duas matrizes com o resultado sendo armazenado na matriz que está associada ao método (**[A]*=[B]** - calcula o produto da matriz **[A]** pela matriz **[B]** e atribui os resultados à matriz **[A]**).

5.5 - AS CLASSES TNode E TNode3d

A classe **TNode**, cujo arquivo de cabeçalho está em parte transcrito a seguir, é uma classe que representa um nó geométrico no espaço bidimensional e serve como classe de base para a derivação outra classe: **TNode3d**, cujos objetos (nós geométricos no espaço tridimensional), possuem dados e métodos comuns que podem e devem ser herdados da classe de base. Essa herança também poderá ocorrer com objetos semelhantes pertencentes a classes que venham a ser implementadas em um outro trabalho por terem alguma característica especial.

Os nós geométricos, bem como as cargas concentradas aplicadas, recalques e restrições aos deslocamentos devem estar referidos no sistema global de coordenadas e a quantidade de nós deve ser no mínimo a necessária para descrever a geometria da estrutura. Assim, deverão ser colocados nós nos seguintes pontos:

- em todos os apoios da estrutura;
- nos locais de aplicação de qualquer tipo de carga concentrada;
- nos locais onde existirem descontinuidades;
- onde houver mudança de alguma propriedade da seção transversal do elemento ou mudança do tipo de material deste;
- onde se desejar saber o valor dos deslocamentos.

Em sua área **protected, TNode** contém os dados mínimos que caracterizam qualquer nó geométrico: as coordenadas X e Y do objeto, bem como seu número (definidos nessa área para que as classes derivadas tenham acesso a eles). Esses dados devem ser fornecidos nessa ordem para a função construtora quando da inicialização de um objeto dessa classe.

Note-se que alguns dos métodos são declarados como virtuais. Esse procedimento permite que a um ponteiro relacionado com o tipo **TNode** seja atribuido o endereço de qualquer objeto de uma das suas subclasses (neste trabalho, um objeto do tipo **TNode3d**), como mostrado pela figura 5.3. Deste modo, na chamada de um dos métodos declarados como virtuais, será executado aquele que estiver relacionado ao tipo de objeto endereçado pelo ponteiro, num claro exemplo do polimorfismo.

Listagem 5. 7 - Parte do arquivo de cabeçalho da classe **TNode**

```
                                    class TNode

{
 protected :
        float coordx, coordy ;              // variaveis para guardar as coordenadas  x e y do node
        unsigned int node_number ;          // variavel para guarda o numero do node
 public :
        TNode ( float X, float Y, unsigned int number ) ;
        virtual ~TNode ( ) { } ;
        float Get_X ( ) { return (coordx) ; }          // retorna o valor da coordenada X do node.
        float Get_Y ( ) { return (coordy) ; }          // retorna o valor da coordenada Y do node.
        unsigned int Get_Number ( ) { return (node_number) ;}      // retorna o numero do node.
        void Chang_Number ( unsigned int new_number=0 ) ; // altera o numero do node.
        virtual void Chang_Coord (AXIS axis, float new_coord) ;     // altera o  valor da
                                           // coordenada axis pelo novo valor  new_coord.
        virtual void Show_Node( ) ;  // exibe o objeto node na tela.
        virtual float Get_Z ( )      { error("sistema de eixos invalido.\n",SAIR) ;
                                       return (0.0) ; }  // usado para evitar aviso de Warning
} ;
```

O método **Chang_Coord** é declarado como virtual porque na implementação deste método é feita uma checagem no valor que foi passado pela variável **axis** (um tipo **enum** declarado no início do arquivo de cabeçalho que pode receber os valores { **x, y, z, xy, xz, yz, nihil** }), havendo uma diferença de procedimento do método conforme o tipo de objeto ao qual está relacionado. Caso a variável **axis** passe os valores **z**, **xz**, **yz** ou **nihil** quando estiver relacionada com um objeto **TNode** ou o valor **nihil** ao se relacionar com um objeto **TNode3d**, uma mensagem de erro será exibida e a execução do programa será encerrada.

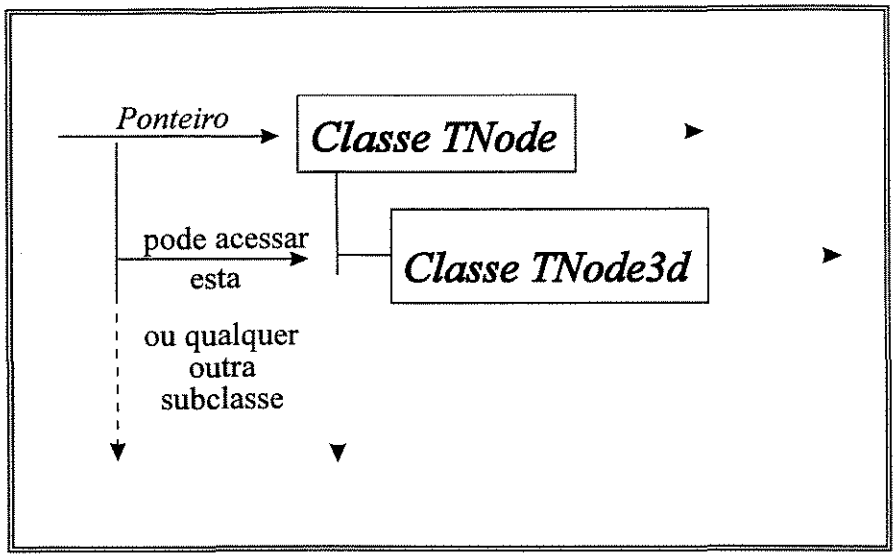

figura 5.3 - Relacionamento entre a classe **TNode** e a classe derivada **TNode3d**

Como mostra parte do arquivo de cabeçalho transcrito à seguir, **TNode3d** é declarada como **public** da classe **TNode**. Sendo assim, herda os dados e os métodos declarados nas áreas **protected** e **public** desta.

Listagem 5. 8 - Parte do arquivo de cabeçalho da classe **TNode3d**

```
                                class TNode3D : public TNode
{
        protected :
                float coordz ;    // variavel para guardar a coordenada z do node

        public :
                TNode3D ( float X, float Y, float Z, unsigned int number ) ;          // construtora
                virtual ~TNode3D ( ) { } ;                                            // destrutora
                virtual float Get_Z ( ) { return (coordz) ; }   // retorna o valor da coordenada Z do
                                                                // node.
                virtual void Chang_Coord (AXIS axis, float new_coord) ;    // altera o valor da
                        // coordenada passada pela variavel axis. O novo valor sera new_coord.
                virtual void Show_Node( ) ;          // exibe o objeto node na tela

};
```

A função construtora de **TNode3d** requer os mesmos dados que os necessários para a função construtora da classe base acrescido da coordenada **Z** (para representar um nó geométrico no espaço tridimensional), porém na sequência de entrada: coordenada **X**, coordenada **Y**, coordenada **Z**, número do node.

Deve-se ressaltar que para objetos da classe **TNode** as funções que recebem como parâmetro uma variável do tipo **enum AXIS** retornarão mensagens de erro e abortarão a execução caso essa variável assuma qualquer um dos valores: **z**, **xz** ou **yz**.

5.6 - AS RESTRIÇÕES AOS DESLOCAMENTOS

Na prática a maioria dos nós de uma estrutura estão com suas vinculações livres em todos os eixos coordenados. Neste projeto considera-se este fato armazenando em uma lista apenas os nós que possuem restrições a deslocamentos e quais as condições dessas restrições (livre para deslocar ou restrito em determinada direção). Assim, os nós (objetos **TNode** ou **TNode3d**) que não estiverem relacionados nessa lista, têm todas as possibilidades de deslocamentos livres e aqueles que estiverem relacionados têm pelo menos uma possibilidade restrita. Com essa metodologia obtém-se duas vantagens:

1. Utilização de uma quantidade de memória menor visto que os nós que não possuem restrição ao deslocamento não reservam espaço para qualquer variável que represente essa restrição;
2. Um menor tempo de processamento quando o sistema avaliar a influência das restrições na análise do problema, pois percorrerá uma lista com alguns poucos nós sujeitos às restrições, não tendo que percorrer toda a lista de nós, onde a grande maioria não esta associada a uma restrição ao deslocamento.

A classe responsável pela criação de um objeto que representa uma situação de restrição em qualquer direção é a classe **TRestricao**, cujo arquivo de cabeçalho está em parte transcrito a seguir.

Listagem 5. 9 - Parte do arquivo de cabeçalho da classe **TRestricao**

```
                                class TRestricao

{
protected :
        unsigned int number_of_node ;
        freedom dof ;           // armazena o grau de liberdade do node em relacao a translacao
                                // em X, Y e Z e a rotacao no plano XY, XZ e YZ
                                // -- 0 para livre e 1 para restrito.

public :
        TRestricao ( unsigned int node_number, unsigned int free_x , unsigned int free_y ,
                unsigned int free_xy ) ;    // construtora quando o node for bidimensional
        TRestricao ( unsigned int node_number, unsigned int free_x , unsigned int free_y ,
                unsigned int free_z , unsigned int free_xy, unsigned int free_xz,
                unsigned int free_yz ) ;    // construtora quando o node for tridimensional
        ~TRestricao ( ) { } ;

        unsigned int Get_Node_Number ( ) { return (number_of_node) ; }   // retorna o numero
                                                                         // do node que tem a restricao
        void Chang_Node_Number ( unsigned int new_number=0 ) ;           // altera o numero do
                                                                         // node que tem a restricao
        void Chang_Restricao(AXIS axis) ;       // altera a restricao no eixo passado como
                                // parametro. Se esta livre, restringe. Se esta restrito, libera.
        unsigned int Get_Restricao(AXIS axis) ;          // retorna a condicao do grau de
                                                         // liberdade no eixo passado como parametro.
        void Get_All_Restricao( freedom & restr ) ;      // retorna a condicao de todas as
                                                         // restricoes que estao atuando no node.
} ;
```

Note-se que existem duas funções construtoras: uma para inicializar um objeto **TRestricao** relacionado com um nó bidimensional, outra para um mesmo objeto porém relacionado com um nó tridimensional. No primeiro caso os parâmetros a serem passados são: número do nó, possibilidade de deslocamento na direção do eixo **X** (translação), possibilidade de deslocamento na direção do eixo **Y** (translação), possibilidade de deslocamento no plano **XY** (rotação), nessa ordem. No segundo caso, os parâmetros são: número do nó seguido das possibilidades de deslocamentos nas direções **X**, **Y**, **Z**, **XY**, **XZ** e **YZ**, nessa ordem. Assim sendo, a

função construtora que será executada dependerá da quantidade de parâmetros passados e estes estão vinculados ao tipo de nó ao qual se deseja atribuir restrição.

Deve-se ressaltar também, a forma de armazenamento dos graus de liberdade de um nó estrutural. Utilizando-se de um recurso do C++ dentro de uma estrutura de dados (*struct*) chamada **freedom**, apresentada na tabela 5.1, trabalha-se *bit* a *bit* para armazenar valores 0 (livre) ou 1 (restrito), representando, assim, os graus de liberdade, ao mesmo tempo que obtém-se um melhor aproveitamento da memória do sistema.

Tabela 5.1 - *Struct* **freedom**

```
struct freedom
  {
        unsigned tranx  : 1 ;   // translacao em x
        unsigned        : 1 ;   //not used
        unsigned trany  : 1 ;   // translacao em y
        unsigned        : 1 ;   //not used
        unsigned tranz  : 1 ;   // translacao em z
        unsigned        : 1 ;   //not used
        unsigned rotxy  : 1 ;   // rotacao em xy
        unsigned        : 1 ;   //not used
        unsigned rotxz  : 1 ;   // rotacao em xz
        unsigned        : 1 ;   //not used
        unsigned rotyz  : 1 ;   // rotacao em yz
        unsigned        : 5 ;   //not used
  } ;
```

A variável **AXIS** presente nas funções ***Chang_Restricao, Get_Restricao*** e ***Get_All_Restricao*** é do tipo **enum** e pode assumir qualquer um dos valores **{ x, y, z, xy, xz, yz, nihil }**, sendo, portanto, do mesmo tipo definido na classe **TNode**.

5.7 - OS RECALQUES

A existência de nós com recalque é, normalmente, menor que a quantidade total de nós, portanto, o procedimento utilizado na classe **TRestricao** de armazenar em uma lista apenas os números dos nós e as restrições nele impostas (ou seja, um objeto **TRestricao**) é aqui repetido, criando-se uma lista onde serão armazenados os números dos nós que tenham recalques, bem como o valor desses recalques .

Assim, para possibilitar a adição de recalques nos nós do problema analisado implementou-se a classe **TRecalque** a qual contém na área **protected** duas variáveis: um *unsigned int* denominado **number_node** que registra o número do nó que apresenta algum recalque e um ponteiro para um objeto da classe **TVetFloat**, de nome **fvet_delta**, que endereça um "vetor" onde estão armazenados os recalques atuantes no nó segundo o esquema da tabela 5.2.

Tabela 5.2 - Posição dos recalques no objeto **fvet_delta** da classe **TVetFloat**

Para um nó bidimensional:	Para um nó tridimensional:
posição [0] --- recalque na direção do eixo X posição [1] --- recalque na direção do eixo Y posição [2] --- recalque no plano XY	posição [0] --- recalque na direção do eixo X posição [1] --- recalque na direção do eixo Y posição [2] --- recalque na direção do eixo Z posição [3] --- recalque no plano XY posição [4] --- recalque no plano XZ posição [5] --- recalque no plano YZ

Com relação ao ponteiro passado na inicialização de um objeto dessa classe deve-se observar a necessidade de mantê-lo coerente com o plano ou espaço da análise. Assim, antes de ser utilizado na chamada ao método construtor de um objeto **TRecalque**, esse ponteiro deverá ser inicializado como segue:

- Para um objeto **TNode** (análise bidimensional) a sintaxe deve ser

ponteiro_para_TVetFloat = new TVetFloat (3)

- Para um objeto **TNode3d** (análise tridimensional) a sintaxe deve ser

ponteiro_para_TVetFloat = new TVetFloat (6)

Com esse ponteiro inicializado, o próximo passo é preencher a variável por ele endereçada com os valores dos recalques nas posições próprias segundo o esquema apresentado na tabela 5.2 (utilizando-se o método **Put_Element** da classe **TVetFloat**), após o que, pode-se criar um objeto **TRecalque** passando-se o endereço contido nesse ponteiro com a seguinte sintaxe

objeto_TRecalque = new TRecalque (número_do_node, ponteiro_para_TVetFloat)

A função construtora fará, então, uma cópia dos valores dos recalques endereçados por ***ponteiro_para_TVetFloat*** na variável endereçada por **fvet_delta**.

Conforme as declarações dos métodos mostradas no arquivo de cabeçalho transcrito a seguir, tem-se que o método **Get_Recalque** retorna o valor do recalque existente no eixo ou plano referido pela variável **axis**, enquanto o método **Chang_Recalque** altera o valor do recalque existente no eixo ou plano passado por **axis**, substituindo-o pelo novo valor que deve ser fornecido através da variável **new_recalq**. Nos dois casos a variável **axis** é do mesmo tipo **enum** que a referida nas classes **TNode** e **TRestricao**. Caso a variável **axis** não seja compatível com o tamanho do "vetor" de recalques atuantes no nó (variável **fvet_delta**), uma mensagem de erro será exibida e a execução do programa interrompida.

Listagem 5. 10 - Parte do arquivo de cabeçalho da classe **TRecalque**

```
                                        class TRecalque
{
 protected :
        TVetFloat * fvet_delta ;
        unsigned int number_node ;

 public :
        TRecalque (unsigned int node_number, TVetFloat * fvet_recalq=NULL);     // construtora
        ~TRecalque ( ) ;                                                        // destrutora

        unsigned int Get_Number_Node ( ) { return (number_node) ; }     // retorna o numero
                        // do node que possui recalque em qualquer um dos eixos ou planos
        float Get_Recalque (AXIS axis) ;        // retorna o valor do recalque que existe no eixo
                                                // ou plano axis
        void Get_All_Recalque (TVetFloat * fvet_recalq=NULL) ;          // retorna todos os
                                                // recalques que existem atuando no node
        void Chang_Recalque(AXIS axis, float new_recalq=0.0) ;          // altera o valor do
                        // recalque que existe no eixo ou plano axis pelo novo valor new_recalq
};
```

O método **Get_All_Recalque** retorna, seguindo o esquema da tabela 5.2, nas posições da variável cujo endereço é passado pelo ponteiro que chama o método (ponteiro para um objeto **TVetFloat**), os valores de todos os recalques atuantes no nó. Esse ponteiro deverá ter tamanho 3 (bidimensional) ou 6 (tridimensional), caso contrário uma mensagem de erro será exibida e a execução do progama interrompida.

5.8 - OS ELEMENTOS

5.8.1- O ELEMENTO GENÉRICO

Pretende-se que cresça o trabalho aqui desenvolvido. Com isso, muito embora nesta etapa tenham sido desenvolvidos apenas elementos de barra com dois nós para tratar o problema a ser analisado, espera-se que em outros projetos venham a ser desenvolvidas classes especializadas que trabalhem com outros tipos de elementos (triangulares, quadriláteros, barras considerando-se não linearidade, análide dinâmica, etc), utilizando-se das propriedades da herança e do polimorfismo.

Tendo-se esse objetivo, propõe-se o esquema da figura 5.4 para a hierarquia das classes dos elementos. Esse esquema é semelhante ao proposto entre as classes de nós geométricos, porém com significativas diferenças em sua implementação. Como mostra o diagrama, pode-se declarar um ponteiro para a classe **TElement** e inicializá-lo com um objeto **TBarra**, **TBarra3D** ou qualquer outro que venha a ser desenvolvido.

Nesta implementação, a classe **TElement** não é meramente uma classe abstrata; sua função construtora gerencia o número do elemento (variável de nome **number_of_element** do tipo *unsigned int* localizada na área **protected**), dado que qualquer objeto de classe derivada contem, devendo-se, portanto, fornecer este valor quando da criação de um objeto desta classe ou de qualquer das suas sub-classes. Na declaração da função construtora existe um valor *default* igual a zero o qual, no corpo da função, é fornecido para esta variável. Deste modo, caso não haja passagem de nenhum valor para a função construtora quando de sua chamada, esta exibirá uma mensagem solicitando ao usuário que dê entrada com o número do elemento pelo teclado.

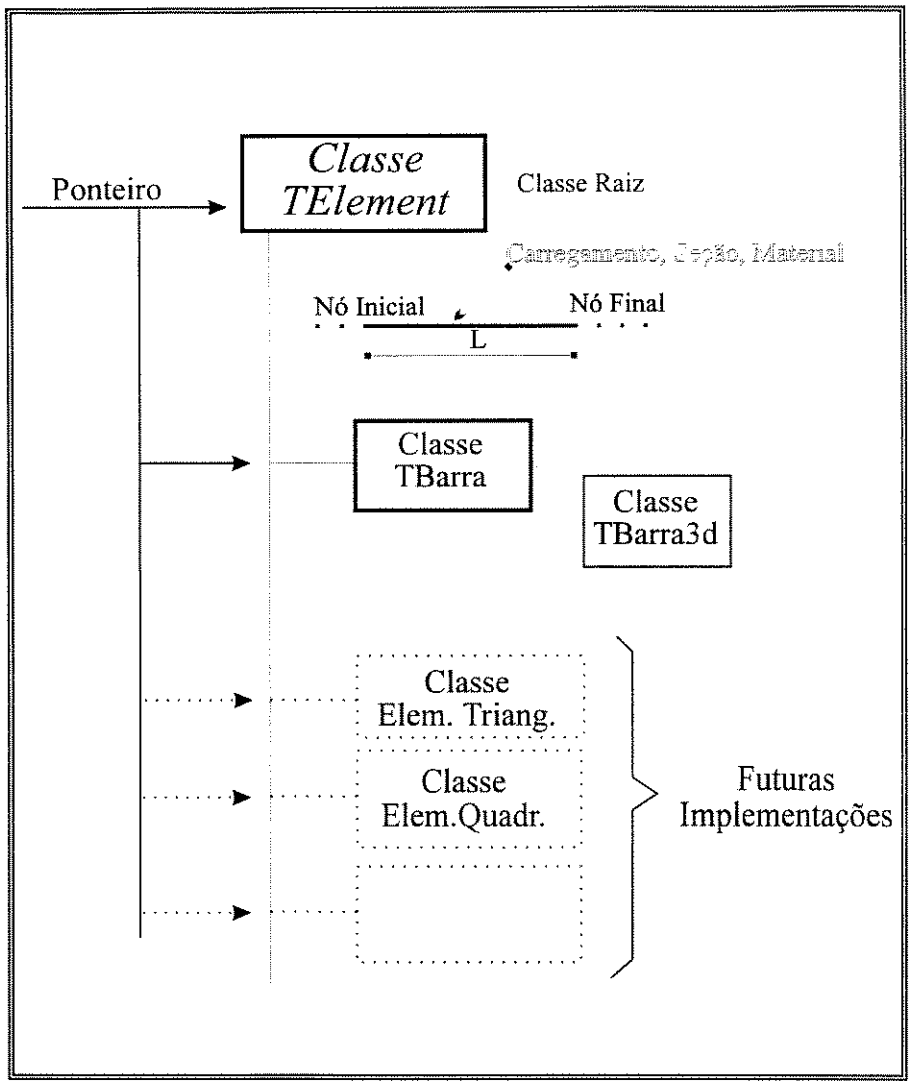

Figura 5.4 - Hierarquia entre a classe TElement e suas classes derivadas

Outra variável que um objeto dessa classe ou de uma das suas sub-classes contém é um ponteiro para uma lista de nós (variável denominada **flistnode**). A atribuição de um endereço para esse ponteiro deve ser feita na inicialização de um objeto de uma das sub-classes mais específicas, pois a quantidade de nós (objetos **TNode** ou **TNode3d**) por elemento pode variar: elementos de barra com 2 ou 3 nós (para levar em consideração variação da seção ao longo do comprimento, por exemplo), elementos triangulares com 3, 6 ou 9 nós, etc. Assim, na inicialização deverá ser passado o endereço de uma lista previamente criada e preenchida com os nós pertencentes ao elemento que será inicializado.

Com relação à lista de nós do elemento (**flistnode**), quando associada a um elemento de barra, deve-se ter em consideração que o primeiro ponteiro armazenado nessa lista representa o nó inicial da barra e o último ponteiro representa o nó final. Neste trabalho essa observação é irrelevante visto que todos os elementos de barra aqui desenvolvidos possuem apenas dois nós, porém em um trabalho futuro onde seja desenvolvida uma classe para representar um elemento de barra com três ou mais nós deve-se ficar atento à esse detalhe, pois os métodos **Get_Size**, **Get_Nf** e **Chang_Nf**, aqui desenvolvidos para objetos de barra em duas e três dimensões, adotam essa convenção.

Listagem 5. 11 - Parte do arquivo de cabeçalho da classe **TElement**

```
                                class TElement
{
 protected:
        unsigned int number ; // variavel que guardar o numero do elemento.
        TList * flistnode ;     // guarda a lista de nodes do elemento

 public :
        TElement ( TList * felement_list_node, unsigned int number_of_element=0 ) ;
        virtual ~TElement ( ) ;

        unsigned int Get_Number ( ) { return (number) ; }
        void Chang_Number (unsigned int new_number=0) ;
        TList * Get_List_Node( ) { return(flistnode) ; }
} ;
```

Os métodos comuns a todos os elementos dessa classe ou dela derivados são **Get_Number** (retorna o valor do número do elemento), **Chang_Number** que altera o número do elemento substituindo-o pelo valor que é passado para a função (caso não seja passado nemhum valor, o método pede que o usuário forneça um número pelo teclado em tempo de execução) e é utilizada quando se retira elementos da estrutura, e **Get_List_Node** (retorna um ponteiro para a.lista de nós do elemento), utilizada para montagem da matriz de rigidez da estrutura.

5.8.2- O ELEMENTO DE BARRA

Derivada da classe **TElement**, a classe **TBarra** guarda as variáveis e propriedades comuns às barras em 2 e 3 dimensões. Sua função construtora recebe como parâmetros um ponteiro para a lista de nós do elemento barra e o número do elemento (se não fornecido é adotado o valor *default* zero), os quais são passados para a construtora da classe **TElement**. Outros dois parâmetros a serem fornecidos à função construtora são um ponteiro para um objeto da classe **TGrupSecao** (indica qual grupo de seção transversal a barra pertence) e um ponteiro para um objeto da classe **TMaterial** (indicativo do tipo de material da barra), ambos declarados com valor *default* **NULL**. Caso não sejam fornecidos qualquer um desses ponteiros a construtora exibirá uma mensagem de erro e encerrará a execução do programa.

Localiza-se na área **protected** as variáveis **size** (do tipo *float*) que armazena o comprimento do elemento barra, **fgrupsecao** (um ponteiro para a classe **TGrupSecao**) e **fmaterial** (um ponteiro para a classe **TMaterial** - classe que descreve o comportamento do material elástico linerar neste trabalho). Esses ponteiros recebem os endereços passados pela função construtora através dos ponteiros **fgrup_of_secao** e **fmat** respectivamente (ver declaração do método no arquivo de cabeçalho transcrito à seguir). Também localizada na área **protected** está um método denominado **Set_Size** declarado como **virtual**, cuja função é a de calcular o comprimento da barra e atribuir o resultado à variável **size**. Sua chamada é feita dentro da função construtora da classe.

Listagem 5. 12 - Parte do arquivo de cabeçalho da classe **TBarra**

```
#define MATBAR2D_SIZE 6        // macro utilizada para determinar a ordem da matriz de rigidez
                               // de um objeto barra com dois nodes e no espaco bidimensional.
                               class TBarra : public TElement
{
 protected :
        TGrupSecao * fgrupsecao ;      // aponta para o tipo de secao transversal da barra.
        TMaterial * fmaterial ;        // guarda qual o tipo de material da barra.
        float size ;                   // guarda o comprimento da barra.
        virtual void Set_Size( ) ;     // metodo que calcula o comprimento da barra e atribui o
                                       // valor a variavel size
 public :
        TBarra ( TList * felement_list_node, unsigned int element_number=0,
                 TGrupSecao * fgrup_of_secao=NULL, TMaterial * fmat = NULL ) ;
                                                              // construtora
        virtual ~TBarra ( ) { } ;              // destrutora
        float Get_Size ( ) { return (size) ; }         // retorna o comprimento da barra
        TMaterial * Get_Material ( ) { return (fmaterial) ; }        // retorna um ponteiro para o
                                                              // objeto TMaterial da barra
        float Get_Modulo_Elasticidade( ) {return(fmaterial->Get_Elasticidade( ));}  // retorna
                                                          // o modulo de elasticidade da barra
        void Chang_Material ( TMaterial * fnew_material=NULL ) ;        // altera o tipo de
                                                              // material da barra
        TSecao * Get_Secao ( ) { return ( fgrupsecao->Get_Secao( ) ) ; }     // retorna um ponteiro
                                                          // para um objeto Tsecao da barra
        void Chang_Grup_Secao (TGrupSecao * fnew_grup=NULL) ;       // altera a qual grupo
                                                          // de secao a barra pertence
        void Chang_Ni (TNode * fnew_ni) ;       // altera o nó inicial da barra
        void Chang_Nf (TNode * fnew_nf) ;       // altera o nó final da barra
        TNode * Get_Ni ( ) { return ( (TNode *)flistnode->Get_Data(1) ) ; }  // retorna um
                    // ponteiro para o objeto TNode que representa o nó inicial da barra
        TNode * Get_Nf ( ) { return((TNode *)flistnode->Get_Data(flistnode->Get_Used())) ; }
              // retorna um ponteiro para o objeto TNode que representa o nó final da barra
                 ////////=========FUNCOES VIRTUAIS=========////////
        virtual void Get_Rig_Matrix_At_Local_Coord ( TSqMatrix * frig_bar ) ;     // calcula a
                    // matriz de rigidez da barra no sistema de eixos local e retorna-a atraves
                    // do ponteiro frig_bar. Essa matriz e de ordem igual a macro
                    // MATBAR2D_SIZE.
        virtual void Get_Rotacional_Matrix ( TSqMatrix * frot_matrix ) ;          // calcula a
                    // matriz de rotacao do sistema de eixos local para o global e retorna-a
                    // atraves do ponteiro frot_matrix. Essa matriz e de ordem igual a macro
                    // MATBAR2D_SIZE.
        virtual void Get_Rig_Matrix_At_Global_Coord ( TSqMatrix *frig_bar ) ;     // calcula a
                    // matriz de rigidez da barra no sistema de eixos global e retorna-a atraves
                    // do ponteiro frig_bar. Essa matriz e de ordem igual a macro
                    // MATBAR2D_SIZE.
    ///////=====metodos virtuais validos apenas para objetos barra tridimensionais=====///////
        virtual float Get_Angulo ( ) ;     // retorna o angulo existente entre o eixo Y da secao
                    // (local) e um eixo global. Exibe mensagem de erro se utilizado com
                    // objetos TBarra e encerra  a execucao do programa.
        virtual void Chang_Angulo( float new_angulo=0 ) ;         // altera o angulo existente
              // entre o eixo Y da secao (local) e um eixo global. Exibe mensagem de erro se
              // utilizado com objetos TBarra e encerra  a execucao do programa.
} ;
```

Na transcrição de parte do arquivo de cabeçalho da classe **TBarra** os métodos são explicados ao lado da sua declaração (alguns deles implementados "*in line*"). Deve-se salientar que as funções **Get_Angulo** e **Chang_Angulo** exibem mensagem de erro e encerram a execução se forem utilizadas com objetos dessa classe, pois estão relacionadas com um ângulo inexistente nas análises com objeto **TBarra** (ver figura 5.5 - ângulo de rotação em torno do eixo X_m).

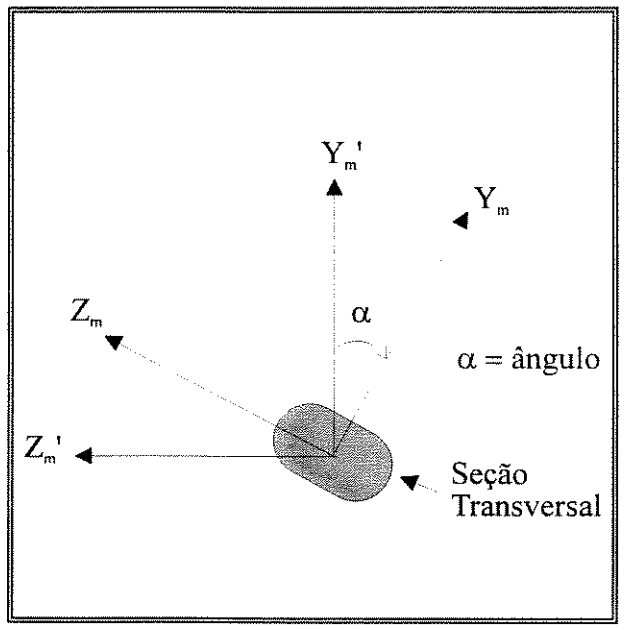

Figura 5.5 - Ângulo de rotação em torno do eixo local X_m

A classe **TBarra3d** é declarada como **public** da classe **TBarra** sendo, portanto, derivada desta e herdando, assim, as variáveis e métodos declarados nas áreas **protected** e **public**. Dessa forma, acrescenta-se apenas uma variável à classe **TBarra3d** (um *float* denominado **axis_angulo** para representar o ângulo α mostrado na figura 5.5 e que deve ser fornecido em radianos) e reescreve-se apenas os métodos cujas rotinas de cálculos são diferentes das realizadas na análise no plano para que seja possível representar um elemento de barra no espaço tridimensional, numa clara vantagem apresentada pela propriedade da herança.

Listagem 5. 13 - Parte do arquivo de cabeçalho da classe **TBarra3d**

```
#define MATBAR3D_SIZE 12      // macro  utilizada para determinar a ordem da matriz de rigidez
                              // de um objeto barra com dois nodes e no espaco tridimensional.

                              class TBarra3d : public TBarra
{
 protected :
        float axis_angulo ;       // guarda o angulo de rotacao do eixo X da secao.
        virtual void Set_Size ( ) ;          // metodo para calcular o comprimento da barra e
                                             // atribuir o resultado a variavel size
 public :
        TBarra3d ( TList * felement_list_node, unsigned int element_number,
                   TGrupSecao * fgrup_of_secao=NULL, TMaterial * fmat = NULL,
                   float angulo=0.0 ) ;                    // construtora
        virtual ~TBarra3d ( ) { } ;                        // destrutora

                        //////////=======FUNCOES VIRTUAIS=======//////////

        virtual void Get_Rig_Matrix_At_Local_Coord ( TSqMatrix * frig_bar ) ;
        virtual void Get_Rotacional_Matrix ( TSqMatrix * frot_matrix ) ;
        virtual void Get_Rig_Matrix_At_Global_Coord ( TSqMatrix *frig_bar ) ;

    ////////=======metodos virtuais validos apenas para objetos barra tridimensionais======//////////

        virtual float Get_Angulo ( ) { return(axis_angulo) ; }      // retorna o angulo existente
                                             // entre o eixo Y da secao (local) e um eixo global.
        virtual void Chang_Angulo( float new_angulo=0 ) ;       // altera o angulo existente
                                             // entre o eixo Y da secao (local) e um eixo global.
} ;
```

Assim, são reescritos os métodos **Get_Rotacional_Matrix**, **Get_Rig_Matrix_At_Local_Coord**, **Get_Rig_Matrix_At_Global_Coord**, **Get_Angulo**, **Chang_Angulo** e **Set_Size**, bem como a função construtora, na qual passa a existir uma variável do tipo *float* chamada **angulo** que passa para o método o valor do ângulo (em radianos) de rotação do eixo X da seção. Essa variável é declarada com o valor *default* zero refletindo o que é mais comum na prática: os elementos não estão com suas seções transversais rotacionadas; e seu valor é atribuido à variável **axis_angulo** da classe.

Para **TBarra3d**, o método **Chang_Angulo** (altera o ângulo de rotação do eixo X local) é declarado com o valor *default* zero para o parâmetro que lhe é passado significando que se nenhum valor for passado, a seção não estará

rotacionada em relação ao sistema global. O valor desse parâmetro deve ser fornecido em radianos e é atribuido à variável **axis_angulo** na implementação do método.

Tanto para objetos **TBarra** como para objetos **TBarra3d**, o método **Get_Rig_Matrix_At_Global_Coord** calcula a matrix de rigidez do elemento barra (com dois nós) em termos do sistema de eixos global do problema e a retorna, através do mesmo ponteiro endereçado com um objeto **TSqMatrix** que lhe é passado como parâmetro, para que possa ser feita sua contribuição na matriz de rigidez global. Dentro do seu código, esse método utiliza-se da função **Get_Rig_Matrix_At_Local_Coord** para calcular a matrix de rigidez do elemento barra em termos de seu sistema de eixos local, passando-lhe um ponteiro para um objeto **TSqMatrix** como parâmetro e através do qual se obtém a matriz desejada. Então, conseguindo-se a matrix de rotação de eixos por meio da função **Get_Rotacional_Matrix**, são feitos os cálculos necessários para se considerar a mudança do sistema de eixos do elemento (local) para o sistema de eixos da estrutura (global).

Existe a possibilidade de se alterar todos os dados do objeto barra, dando-se especial atenção a sua seção transversal que pode ser alterada em apenas uma barra - um pilar de um edifício, por exemplo - com a utilização do método **Chang_Grup_Secao** associado à barra que se deseja altera a seção ou para todo um conjunto de barras - banzos de uma treliça, por exemplo - com a utilização do método **Chang_Grup_Secao** associado ao grupo que se deseja alterar a seção (pertencente à classe **TGrupSecao**). Esse recurso é muito útil na prática quando se pretende executar um dimensionamento do elemento, onde não raras são as vezes em que tal cálculo nos leva a conclusão de que a seção não atende aos padrões de resistência ou é por demais resistente, tornando-se anti-econômica. A solução para essas situações é a mudança da seção transversal da peça analisada, daí a importância da existência dessa opção.

5.9 - OS ARQUIVOS DE PERFIS E AS SEÇÕES TRANSVERSAIS

Neste projeto deve-se associar a um elemento de barra (o único aqui tratado) uma determinada seção transversal que em muitos casos (principalmente em estruturas metálicas) existem tabeladas em catálogos dos fabricantes, os quais fornecem as informações necessárias tais como área, dimensões, peso, momentos de inércia, etc.

Planejou-se, então, obter essas informações a partir de arquivos de dados previamente armazenados no disco rígido , sendo, portanto, necessário primeiramente criar tais arquivos.

Com esse objetivo foi implementado um pequeno programa que se utiliza de uma classe raiz abstrata para tratar com arquivos de seções transversais e outras classes derivadas dessa, representando cada uma um arquivo para um determinado tipo de seção transversal. Assim formou-se a hierarquia de classes mostrada na figura 5.6.

A exemplo do procedimento adotado para **TNode** e **TElement**, um ponteiro para um objeto **TArquivos_de_Secoes** pode ser inicializado com qualquer uma das classes derivadas, como se percebe pela figura 5.6. Esse procedimento permitirá aumentar indefinidamente a quantidade de classes derivadas, bastando para isso que cada uma delas seja declarada como **public** da classe raiz e que possua, além das funções construtora e destrutora, as mesmas declarações virtuais seguintes:

```
virtual void Ler_Secao (int num = 0) ;
virtual void Get_New_Secao(void * cant);
virtual void Show_Secao(void * cant);
virtual void Corrigir (int number = 0) ;
virtual void Procurar( void * sec , int & aux );
virtual void * Procurar( float h=0, float b=0, float e=0);
virtual void Escreve_no_Final( );
```

Métodos da classe de TArquivos_de_Secoes e classes derivadas

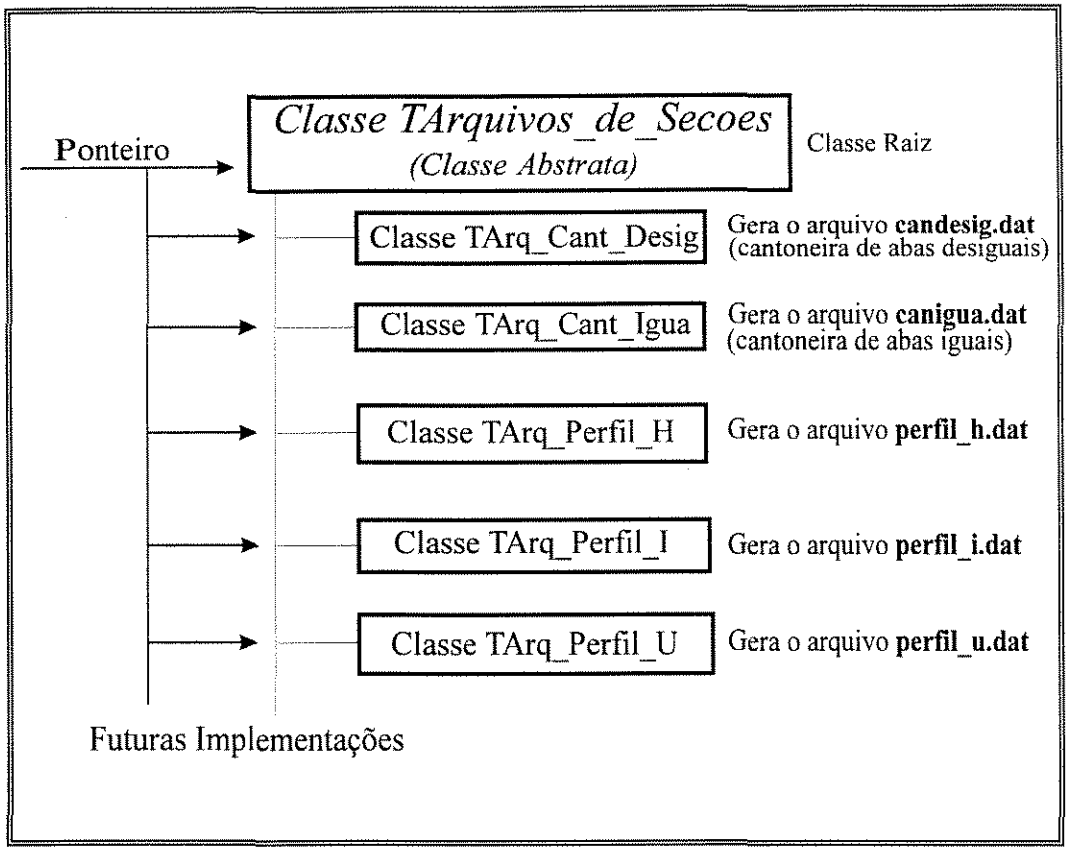

Figura 5.6 - Hierarquia Entre as Classes Que Manipulam Arquivos de Seções

As classes de arquivos não dispõem de variáveis de dados a serem inicializadas, mas se utilizam de uma série de *structs* declaradas no arquivo **my_struct.h**, uma para cada tipo de seção. As funções dessas classes, além da construtora e destrutora que nada executam, são:

```
virtual void Get_New_Secao (void *cant) ;
```

- - método **private** que é usado por outras funções (**Corrigir** e **Escrever**) e cuja função é pedir ao usuário que forneça os dados de uma nova seção;

```
virtual void Show_Secao (void *cant) ;
```

- - método **private** que é usado por outras funções (**Ler_Secao**) e cuja função é exibir na tela os dados de uma seção armazenada;

virtual void Corrigir (int number = 0) ;

- - Tem a função de corrigir qualquer seção (armazenada no arquivo sob número **number**) que tenha tido qualquer um de seus dados gravado erroneamente;

virtual void Escreve_no_Final ( );

- - método que é o responsável pelo aumento da quantidade de seções armazenadas nos arquivos pois grava os dados de uma nova seção no final do arquivo correspondente;

virtual void Procurar(void * sec , int & aux) ;

- Classe **TCantoneira_Igual** : utiliza os métodos da classe **TArq_Cant_Igua** para ler no arquivo **canigua.dat** os dados de uma seção cantoneira de abas iguais, requerida através das variáveis *float* passadas como parâmetros. Caso a seção seja localizada no arquivo, os dados lidos são transcritos em uma variável pertencente a sua área **private** do tipo *struct* **Canto_Igua** . Essa *struct* encontra-se declarada no arquivo **my_struc.h**;

- Classe **TSecao_Perfil_H** : utiliza os métodos da classe **TArq_Perf_H** para ler no arquivo **perfil_h.dat** os dados de uma seção de perfil H, requerida através das variáveis *float* passadas como parâmetros. Caso a seção seja localizada no arquivo, os dados lidos são transcritos em uma variável pertencente a sua área **private** do tipo *struct* **Perfil_H** . Essa *struct* encontra-se declarada no arquivo **my_struc.h**;

- Classe **TSecao_Perfil_I** : utiliza os métodos da classe **TArq_Perf_I** para ler no arquivo **perfil_i.dat** os dados de uma seção de perfil I, requerida através das variáveis *float* passadas como parâmetros. Caso a seção seja localizada no arquivo, os dados lidos são transcritos em uma variável pertencente a sua área **private** do tipo *struct* **Perfil_I** . Essa *struct* encontra-se declarada no arquivo **my_struc.h**;

- Classe **TSecao_Perfil_U** : utiliza os métodos da classe **TArq_Perf_U** para ler no arquivo **perfil_u.dat** os dados de uma seção de perfil U, requerida através das variáveis *float* passadas como parâmetros. Caso a seção seja localizada no arquivo, os dados lidos são transcritos em uma variável pertencente a sua área **private** do tipo *struct* **Perfil_U** . Essa *struct* encontra-se declarada no arquivo **my_struc.h**;

- Classe **TSecao_Qualquer** : não utiliza os métodos para ler de arquivo os dados de uma seção. As variáveis *float* passadas como parâmetros são os dados da seção: área, Jx, Jy, Jt e peso (nessa ordem), os quais são

transcritos pela construtora da classe para uma variável do tipo *struct* **secao_qualquer** existente na área **private**. Essa *struct* encontra-se declarada no arquivo **my_struc.h**;

Os métodos aos quais a classe **TGeneric_Secao** e suas classes derivadas respondem são apresentados à seguir. Na implementação de uma nova classe derivada dessa, deve-se providenciar a declaração e a implementação de uma função de mesmo nome para cada uma das aqui mostradas.

Listagem 5. 14 - Parte do arquivo de cabeçalho da classe **TGeneric_Secao**

```
TGeneric_Secao                  () {};
virtual ~TGeneric_Secao         () {};
virtual float Get_Altura        () =0;
virtual float Get_Aba           () =0;
virtual float Get_Esp           () =0;
virtual float Get_Peso          () =0;
virtual float Get_Area          () =0;
virtual float Get_J_xx          () =0;
virtual float Get_W_xx          () =0;
virtual float Get_I_xx          () =0;
virtual float Get_Y_xx          () =0;
virtual float Get_J_yy          () =0;
virtual float Get_W_yy          () =0;
virtual float Get_I_yy          () =0;
virtual float Get_X_yy          () =0;
virtual float Get_Jmax_11       () =0;
virtual float Get_Imax_11       () =0;
virtual float Get_Jmin_22       () =0;
virtual float Get_Imin_22       () =0;
virtual float Get_Tang_alfa_y_22 () =0;
virtual float Get_Base          () =0;
virtual float Get_W_pl_xx       () =0;
virtual float Get_W_pl_yy       () =0;
virtual float Get_Esp_Alma      () =0;
virtual float Get_Esp_Mesa      () =0;
virtual float Get_Jtorcao       () =0;
virtual void Show_Secao         () =0;
```

Dessa forma resolveu-se o problema de representar uma seção transversal restando outro: atribuir uma seção transversal à uma barra.

A idéia natural de atribuir o endereço de um objeto da classe **TGeneric_Secao** à uma variável tipo ponteiro associado ao objeto da classe **TBarra** esbarra em um possível problema: como da classe **TGeneric_Secao** podem derivar

outras classes não incluidas neste trabalho, a criação de uma nova classe de seção transversal implicaria em

- inserir essa nova opção na função que executa a inicialização do ponteiro da classe **TBarra** para as seções e
- incluir essa opção na função que faz a leitura do arquivo de dados,

ou seja, haveria a necessidade de se alterar código já testado.

O problema foi contornado (ou minimizado) com a criação de uma nova classe chamada **TSecao** que tem em sua área **protected** um ponteiro para um objeto **TGeneric_Secao** e uma variável do tipo *char (*denominada **format***)*. Sua função construtora, transcrita à seguir, recebe como um dos parâmetros um caracter e atribui-o a essa variável sendo esse *char* responsável pela inicialização do ponteiro para o objeto da classe **TGeneric_Secao** através do comando ***switch***.

Listagem 5. 15 - Método construtor da classe **TSecao**

```
TSecao :: TSecao ( char A, float B, float C, float D, float E, float F)
{
        format = toupper(A) ;
        switch (format)
                {
                 case 'C' : fptrsec = new TCantoneira_Igual(B,C,D,E,F) ; break ;
                 case 'L' : fptrsec = new TCantoneira_Desigual(B,C,D,E,F) ; break ;
                 case 'H' : fptrsec = new TSecao_Perfil_H(B,C,D,E,F) ; break ;
                 case 'I' : fptrsec = new TSecao_Perfil_I(B,C,D,E,F) ; break ;
                 case 'U' : fptrsec = new TSecao_Perfil_U(B,C,D,E,F) ; break ;
                 case 'Q' : fptrsec = new TSecao_Qualquer(B,C,D,E,F) ; break ;
                 default : error("secao nao implementada em TSecao.\n") ;
                }
}
```

Os outros parâmetros que devem ser fornecidos à construtora são valores do tipo **float** e estão diretamente relacionados ao tipo da seção transversal, sendo eles:

- os valores do comprimento da aba e a espessura da cantoneira quando o perfil for uma cantoneira de abas iguais;
- os valores da altura, o comprimento da aba e a espessura da cantoneira quando o perfil for uma cantoneira de abas desiguais;
- os valores da altura e o peso do perfil quando se tratar de perfis I, H ou U;
- os valores da área, de Jx, Jy, Jt e o peso quando for qualquer outro tipo de seção.

Perceba-se pela construtora que esses valores são utilizados para a inicialização do ponteiro ***fptrsec*** (objeto do tipo **TGeneric_Secao**), variável da classe **TSecao**, e são os mesmos utilizados pelas restectivas classes derivadas de **TArquivos_de_Secoes** para encontrar os dados das seções nos arquivos correspondentes.

Apresenta-se à seguir o arquivo de cabeçalho da classe **TSecao**. A maioria dos métodos não necessitam de parâmetros de entrada e têm como função retornar os valores que seus nomes representam. Note-se o método **Chang_Formato** que recebe como parâmetro os mesmos dados da função construtora. Esse método permite alterar o ponteiro ***fptrsec*** do objeto **TSecao** primeiramente utilizando o comando **delete** para esse ponteiro e inicializando-o novamente de modo idêntico ao utilizado na construtora, representando, assim, a substituição de uma seção transversal por outra.

Listagem 5. 16 - Parte do arquivo de cabeçalho da classe **TSecao**

```
                                        class TSecao
{
 protected :
          TGeneric_Secao *fptrsec ;
          char format ;
 public :
          TSecao ( char A, float B=0, float C=0, float D=0, float E=0, float F=0) ;
          ~TSecao ( ) { delete fptrsec ; } ;
          char Formato ( ) ;
          void Chang_Formato ( char G, float J=0, float L=0, float M=0, float N=0, float P=0 ) ;
          float Altura          ( ) { return ( fptrsec->Get_Altura( ) ) ;          }
          float Aba             ( ) { return ( fptrsec->Get_Aba( ) ) ;             }
          float Esp             ( ) { return ( fptrsec->Get_Esp( ) ) ;             }
          float Peso            ( ) { return ( fptrsec->Get_Peso( ) ) ;            }
          float Area            ( ) { return ( fptrsec->Get_Area( ) ) ;            }
          float J_xx            ( ) { return ( fptrsec->Get_J_xx( ) ) ;            }
          float W_xx            ( ) { return ( fptrsec->Get_W_xx( ) ) ;            }
          float I_xx            ( ) { return ( fptrsec->Get_I_xx( ) ) ;            }
          float Y_xx            ( ) { return ( fptrsec->Get_Y_xx( ) ) ;            }
          float J_yy            ( ) { return ( fptrsec->Get_J_yy( ) ) ;            }
          float W_yy            ( ) { return ( fptrsec->Get_W_yy( ) ) ;            }
          float I_yy            ( ) { return ( fptrsec->Get_I_yy( ) ) ;            }
          float X_yy            ( ) { return ( fptrsec->Get_X_yy( ) ) ;            }
          float Jmax_11         ( ) { return ( fptrsec->Get_Jmax_11( ) ) ;         }
          float Imax_11         ( ) { return ( fptrsec->Get_Imax_11( ) ) ;         }
          float Jmin_22         ( ) { return ( fptrsec->Get_Jmin_22( ) ) ;         }
          float Imin_22         ( ) { return ( fptrsec->Get_Imin_22( ) ) ;         }
          float Tang_alfa_y_22  ( ) { return ( fptrsec->Get_Tang_alfa_y_22( ) ) ;  }
          float Base            ( ) { return ( fptrsec->Get_Base( ) ) ;            }
          float W_pl_xx         ( ) { return ( fptrsec->Get_W_pl_xx( ) ) ;         }
          float W_pl_yy         ( ) { return ( fptrsec->Get_W_pl_yy( ) ) ;         }
          float Esp_Alma        ( ) { return ( fptrsec->Get_Esp_Alma( ) ) ;        }
          float Esp_Mesa        ( ) { return ( fptrsec->Get_Esp_Mesa( ) ) ;        }
          float Get_Jt          ( ) { return ( fptrsec->Get_Jtorcao( ) ) ;         }
          void Show             ( ) { fptrsec->Show_Secao( ) ;                     }
} ;
```

Quando houver acréscimo na quantidade de classes de seções (classes derivadas de **TGeneric_Secao**), a declaração de uma sub-classe de **TSecao** que tenha em sua construtora uma função ***switch*** com as novas opções será suficiente para permitir a inicialização do ponteiro para **TGeneric_Secao** com as novas seções. Essa construtora primeiramente chamará a construtora de **TSecao**, fará a atribuição da variável *char*, após o que comparará essa variável com as opções já existentes, para então, caso não encontre igualdade na comparação, exibirá uma mensagem de erro que deve ser ignorada e iniciará a pesquisa pelas novas opções implementadas.

Poder-se-á, também, acrescentar uma nova linha contendo a nova opção dentro da função ***switch*** da função construtora da classe **TSecao** (uma nova opção ***case***), processo mais prático, porém sujeito a introdução de erro no código estável.

Procurando-se por um melhor gerenciamento da quantidade de memória do sistema e rapidez de processamento, optou-se por não atribuir uma seção transversal à cada um dos elementos diretamente, mas sim, criar uma lista de seções transversais, um objeto da classe **TList**, e atribuir a seção à um grupo de elementos. Dessa forma traduz-se para a área científica dois fatos rotineiros na prática:

- várias peças de uma estrutura têm a mesma seção transversal (pilares e vigas em edifícios por exemplo). Assim, se existirem **n** peças com a mesma seção, ao invés de serem armazenados os dados de **n** seções iguais, armazena-se os dados de uma seção e **n** ponteiros para a sua posição na memória, o que economiza uma considerável quantidade desta.
- no dimensionamento de determinados tipos de estruturas (treliças, por exemplo) pode ocorrer a necessidade da substituição de uma seção transversal comum a vários elementos por uma outra seção que seja igual para todos (como no caso dos banzos). Com o esquema aqui adotado esse procedimento pode ser agilizado, pois pode ser feito com um único comando para todas as barras em substituição ao procedimento de se alterar as seções dos elementos um a um.

Para tanto, implementou-se a classe **TGrup_Secao** que contém em sua área **protected** um ponteiro para um objeto **TSecao** (denominado **fsecao**) e apresenta dois exemplos de polimorfismo: a própria função construtora e o método **Chang_Grup_Secao**.

No primeiro caso, em uma das declarações há a necessidade de se fornecer como parâmetro, através de um ponteiro, o endereço de um objeto **TSecao**, o qual será atribuido à variável da classe **fsecao** na implementação do método, enquanto que, para o mesmo propósito, na outra declaração deve-se fornecer o endereço da lista de seções (variável que será apresentada na classe **TSec_Manager**) assim como o número que indica a posição nessa lista na qual os dados da seção foram armazenados (declarado com o valor *default* zero e se não for fornecido o método solicitará ao usuário que o faça através do teclado em tempo de execução).

Listagem 5. 17 - Parte do arquivo de cabeçalho da classe **TGrup_Secao**

```
                                class TGrup_Secao
{
        protected :
                TSecao * fsecao ;  // ponteiro para a secao

        public :
                TGrup_Secao ( TSecao * fsecao_of_grup ) ;
                TGrup_Secao ( TList * flist_of_sec, unsigned int number_of_position=0 ) ;
                ~TGrup_Secao ( ) { } ;
                void Chang_Grup_Secao ( TSecao * fnew_secao ) ;
                void Chang_Grup_Secao ( TList * flist_of_sec,
                                        unsigned int new_number_of_secao=0) ;
                TSecao * Get_Secao ( ) { return (fsecao) ; }
} ;
```

O segundo exemplo de polimorfismo dessa classe (as duas declarações para o método **Chang_Grup_Secao**), permite ao usuário alterar a seção para a qual a variável da classe **fsecao** "aponta". A sua utilização associado ao objeto **TGrup_Secao** permite alterar a seção transversal de todo um conjunto de elementos (barras) cujas variáveis **fgrupsecao** (ponteiro para um objeto **TGrup_Secao** declarado na área **protected** da classe **TBarra**) enderecem tal objeto **TGrup_Secao**.

O esquema da figura 5.7 apresenta as duas possibilidades de alteração das seções transversais das barras permitidas neste trabalho. As variáveis G1, G2, G3 e G4 são ponteiros **void** endereçados com objetos do tipo **TGrup_Secao**

e armazenados na lista de grupos de seções enquanto S1, S2 e S3 são ponteiros **void** endereçados com objetos do tipo **TSecao** e armazenados na lista de seções. A figura 5.7-a representa o esquema inicial e a figura 5.7-b o esquema após a alteração das seções das barras.

Note-se que as Barras 1 e 2 tinham inicialmente a variável **fgrupsecao** com o endereço do objeto G1 e este tinha sua variável **fsecao** com o endereço do objeto S1 representando, assim, que essas barras têm seções transversais do tipo Seção 1. Na situação final, com a utilização do método **Chang_Grup_Secao** (método da classe **TGrup_Secao**) associado ao objeto G1, altera-se as seções transversais das duas barras com um só comando, pois agora a variável **fsecao** do objeto G1 endereça o objeto S2, representando, assim, que as Barras 1 e 2 têm seções transversais do tipo Seção 2.

Com relação às Barras 3 e 4 o esquema mostra que elas tinham inicialmente a variável **fgrupsecao** com o endereço do objeto G2 e este tinha sua variável **fsecao** com o endereço do objeto S2 representando, assim, que essas barras têm seções transversais do tipo Seção 2. Na situação final, com a utilização do método **Chang_Grup_Secao** (método da classe **TBarra**) associado ao objeto Barra 4, altera-se a seção transversal apenas dessa barra, pois agora sua variável **fgrupsecao** endereça o objeto G3 que por sua vez tem sua variável **fsecao** endereçando o objeto S3, representando, assim, que a Barra 4 tem agora seção transversal do tipo Seção 3.

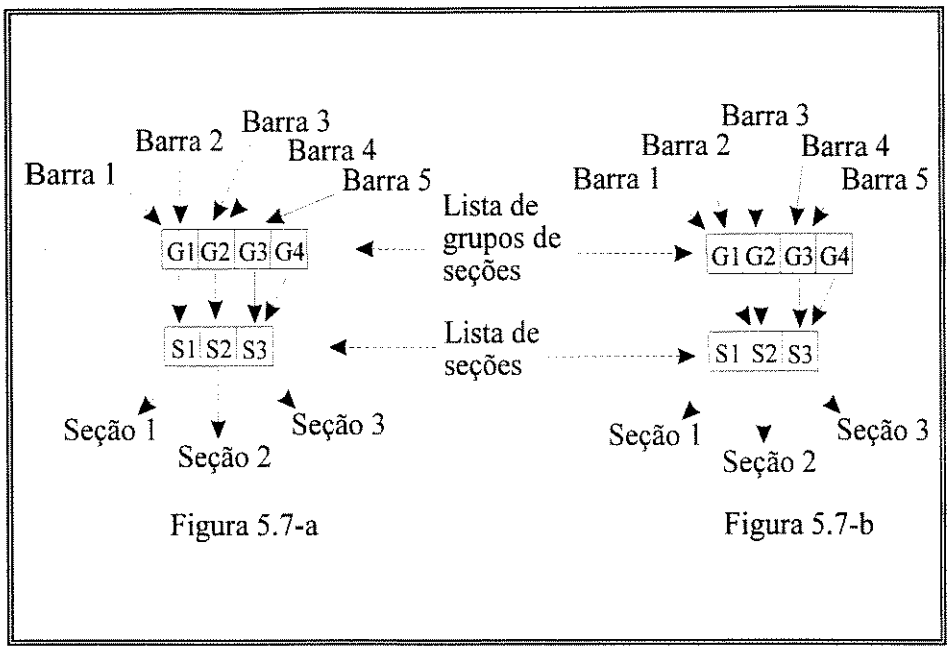

Figura 5.7 - Esquema para atribuições e alterações das seções às barras

5.10 - OS MATERIAIS DOS ELEMENTOS

Os materiais que compõem os elementos devem refletir a relação Tensão x Deformação e são representados, neste projeto e nos que venham a este ser adicionados, por uma classe específica para cada comportamento.

A classe aqui implementada, denominada **TMaterial**, representa o modelo cuja relação entre o estado de tensões e o de deformações se faz de forma linear, portanto **TMaterial** é uma classe que cria objetos representantes do modelo eslásto-linear. Para se introduzir um outro comportamento deve-se declarar uma outra classe derivada desta e que reflita a nova relação (modelo elásto-plástico, por exemplo).

O arquivo de cabeçalho transcrito mostra a existência na área **protected** de uma variável do tipo ***long double*** denominada **modulo** e que armazena o valor do módulo de elasticidade do material, e outra do tipo ***float*** denominada **poisson**

para armazenar o coeficiente de Poisson. Pela função construtora são atribuidos à essas variáveis os valores dos parâmetros **modulo_elasticidade** e **coef_poisson**, respectivamente. Esses parâmetros são declarados com os valores *default* das macros definidas no início do arquivo **MOD_ELASTICIDADE** e **COEF_POISSON**

Listagem 5. 18 - Parte do arquivo de cabeçalho da classe **TMaterial**

```
#define MOD_ELASTICIDADE 210000
#define COEF_POISSON 0.1

                                        class TMaterial
{
 protected :
        long double modulo ;
        float poisson ;

 public :
        TMaterial ( long double modulo_elasticidade = MOD_ELASTICIDADE,
                                        float coef_poisson=COEF_POISSON ) ;   // construtora
        virtual ~TMaterial ( ) { } ;                                          // destrutora
        virtual float Get_Elasticidade( ) { return (modulo) ; }       // retorna o valor do modulo
                                                                      // de elasticidade do material
        virtual float Get_Elasticidade_Transversal( ) ;               // calcula e retorna o valor do
                                                        // modulo de elasticidade transversal do material
        virtual void Chang_Elasticidade(long double new_elasticidade=0.0) ;// altera o valor do
                                                                // modulo de elasticidade do material
        virtual void Chang_Coef_Poisson (float new_coef_poisson=0.0) ;        // retorna o
                                                                // valor do coeficiente de Poisson
};
```

Os métodos **Chang_Elasticidade** e **Chang_Coef_Poisson** permitem substituir os valores das variáveis **modulo** e **poisson** pelos valores dos parâmetros passados para a função. Como na declaração esses parâmetros recebem o valor *default* zero, se não for fornecido um valor diferente na chamada ao método, este solicitará que o usuário o faça através do teclado em tempo de execução. Procedimento semelhante ocorre com a função construtora caso o valor fornecido para qualquer um dos parâmetros na inicialização de um objeto seja zero.

5.11 - CARREGAMENTO : UMA COLETÂNEA DE CLASSES

O carregamento de um modelo (estrutura) é considerado aqui como composto pelas cargas atuantes diretamente sobre os nós e pelas cargas que atuam nos elementos (barras neste trabalho). Para representar essas cargas, tem-se implementadas as classes **TNode_Load** e **TBar_Load**.

Listagem 5. 19 - Parte do arquivo de cabeçalho da classe **TNode_Load**

```
                                        class TNode_Load
{
 protected :
        TVetFloat * fvetload ;
        unsigned int number_node ;

 public :
        TNode_Load (float node_number, TVetFloat * fvetcarg=NULL);
        ~TNode_Load ( ) ;

        unsigned int Get_Number_Node ( ) { return (number_node) ; }      // retorna o numero
                                  // do node sujeito as cargas armazenadas na variavel fvetload
        float Get_Load (AXIS axis) ;      // retorna o valor da carga atuando na direcao AXIS
        void Get_All_Loads (TVetFloat * fvetcarg=NULL) ;          // retorna no vetor passado
                          // como parametro os valores de todas as cargas que atuam sobre o node
        void Chang_Load(AXIS axis, float new_load=0.0) ;// altera o valor da carga que atua
                                                  // na direcao AXIS pelo novo valor new_load
} ;
```

A classe **TNode_Load** contém em sua área **protected** uma variável ponteiro para um objeto **TVetFloat**. Essa variável, denominada **fvetload**, é utilizada para armazenar, segundo o esquema da tabela 5.2, as cargas que atuam sobre um nó. Essa classe também contém na mesma área uma variável do tipo *unsigned int* denominada **number_node** na qual é armazenado o número do nó sujeito às forças contidas em **fvetload**.

Tabela 5.2 - Posição das forças na variável **fvetload**

Para um nó bidimensional:	Para um nó tridimensional:
posição [0] --- força na direção do eixo X posição [1] --- força na direção do eixo Y posição [2] --- momento no plano XY	posição [0] --- força na direção do eixo X posição [1] --- força na direção do eixo Y posição [2] --- força na direção do eixo Z posição [3] --- momento no plano XY posição [4] --- momento no plano XZ posição [5] --- momento no plano YZ

A função construtora da classe recebe como parâmetros o valor do número do nó sobre o qual existem cargas atuantes e o endereço do vetor onde essas cargas estão armazenadas (ponteiro **fvetcarg** declarado com o valor *default* **NULL**). Inicializa-se, então, a variável **fvetload** com a mesma ordem do ponteiro passado como parâmetro através da sintaxe

fvetload = new TVetFloat (fvetcarg->Get_Size()) ;

após o que atribui-se à **fvetload** uma cópia dos valores contidos em **fvetcarg**. Este procedimento permite que a classe seja utilizada para representar cargas em nós bi ou tridimensionais. Caso a ordem do ponteiro **fvetcarg**, conseguida através do método **Get_Size** associado ao objeto, não seja igual à 3 ou 6 (os únicos valores permitidos neste trabalho), o método exibirá uma mensagem de erro e encerará a execução do programa.

Nota-se pela transcrição do arquivo de cabeçalho que os métodos **Get_Load** e **Chang_Load** necessitam como parâmetro de uma variável do tipo *enum* **AXIS**, a mesma já comentada nos métodos das classes **TRestricao** e **TRecalque**.

Para implementar a classe de cargas atuando sobre as barras (classe **TBar_Load**) primeiramente foram desenvolvidas classes que representam apenas os tipos de cargas. Assim, utilizando-se do conceito de herança foi criada uma classe abstracta denominada **TGeneric_Load** da qual todas as outras classes mais específicas são derivadas, como mostra o esquema da figura 5.8.

Listagem 5. 20 - Parte do arquivo de cabeçalho da classe **TGeneric_Load**

```
                                    class TGeneric_Load
{ public :
        TGeneric_Load ( ) { } ;
        virtual ~TGeneric_Load ( ) { } ;
        virtual float Get_M1y ( ) =0 ;
        virtual float Get_M2y ( ) =0 ;
        virtual float Get_M1x ( ) =0 ;
        virtual float Get_M2x ( ) =0 ;
        virtual float Get_M1z ( ) =0 ;
        virtual float Get_M2z ( ) =0 ;
        virtual float Get_F1x ( ) =0 ;
        virtual float Get_F2x ( ) =0 ;
        virtual float Get_F1y ( ) =0 ;
        virtual float Get_F2y ( ) =0 ;
        virtual float Get_F1z ( ) =0 ;
        virtual float Get_F2z ( ) =0 ;
        virtual void Show_Carg ( ) =0 ;
} ;
```

Assim, definindo-se um ponteiro para a classe raiz **TGeneric_Load** pode-se inicializá-lo com o endereço de um objeto pertencente a qualquer uma das classes derivadas. Note-se que para que isso se concretize é necessário que os métodos de todas as sub-classes sejam declarados como virtuais.

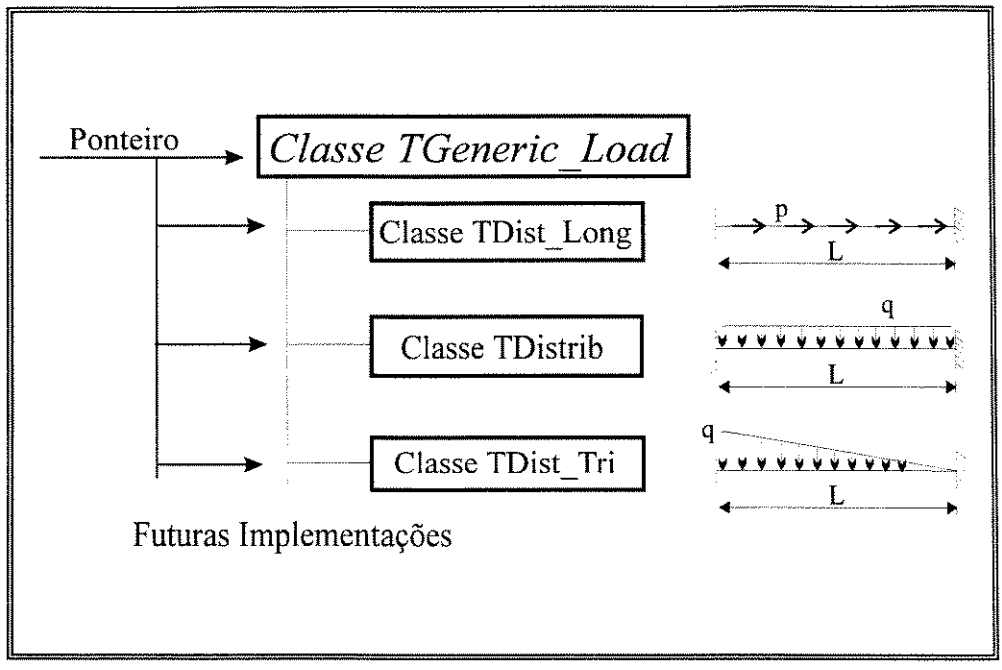

Figura 5.8 - A hierarquia entre as classes de tipos de carregamentos

Como está mostrado na transcrição do arquivo de cabeçalho da classe **TGeneric_Load**, ela é abstracta, ou seja, nenhum objeto de seu tipo pode ser criado e seus métodos nada executam. Essa é uma situação típica (SWAN[1993], PETZOLD[1993], WEISKAMP[1993]) quando se deseja utilizar as vantagens da herança em situações onde são muitas as possibilidades de classes derivadas.

Derivadas da classe **TGeneric_Load** foram implementadas as seguintes sub-classes:

- Classe **TDist_Long** (carga distribuida aplicada longitudinalmente em todo o vão). Necessita como parâmetros de entrada do valor da carga e do comprimento da barra.
- Classe **TDistrib** (carga uniformemente distribuida aplicada normalmente ao longo do vão). Necessita como parâmetros de entrada do valor da carga, do comprimento da barra e, caso a análise seja tridimensional, do ângulo entre a carga e o eixo local Y_m, se for diferente de zero.

- Classe **TDist _Tri** (carga linearmente distribuida aplicada normalmente ao longo do vão). Necessita como parâmetros de entrada do valor máximo da carga, do comprimento da barra e, caso a análise seja tridimensional, do ângulo entre a carga e o eixo local Y_m, se for diferente de zero.

Todas essas classes possuem suas variáveis de dados declaradas na área **private** e têm todas as suas funções declaradas como virtuais, respondendo assim de modo diferente à chamada de uma função de mesmo nome (polimorfismo). A seguir encontra-se o arquivo de cabeçalho da classe **TDistrib** que exemplifica o conjunto das classes derivadas.

Listagem 5. 21 - Parte do arquivo de cabeçalho da classe **TDistrib**

```
                        class TDistrib : public TGeneric_Load
{
 private:
        float q, len, angulo;       // q = valor da carga;       len = comprimento da barra
                                    // angulo = angulo entre a carga e o eixo local Y. OBS.: somente
                                    // deve ser fornecido valor a essa variavel para analise tridimensional,
                                    // caso contrario havera erro nos valores calculados.
 public :
        TDistrib (float carga=0 , float lenght=0 , float ang=0) ;
        virtual ~TDistrib           () {} ;
        virtual float Get_F1x       () { return (0.0) ; }
        virtual float Get_F2x       () { return (0.0) ; }
        virtual float Get_M1x       () { return (0.0) ; }
        virtual float Get_M2x       () { return (0.0) ; }
        virtual float Get_F1y       () ;
        virtual float Get_F2y       () ;
        virtual float Get_M1y       () ;
        virtual float Get_M2y       () ;
        virtual float Get_F1z       () ;
        virtual float Get_F2z       () ;
        virtual float Get_M1z       () ;
        virtual float Get_M2z       () ;
        virtual void Show_Carg      () ;
} ;
```

Note-se que a possibilidade de introdução de novas classes ao sistema está aberta, bastando para isso que a nova classe seja derivada de uma sub-classe já existente ou da classe raiz, devendo apresentar as mesmas declarações de funções virtuais que estas.

A classe **TBar_Load** representa uma carga atuando em uma barra. Para tanto, apresenta em sua área **protected** três variáveis: um ponteiro para um objeto da classe **TGeneric_Load** denominado **fptrcarg**, uma variável do tipo *char* denominada **tip** que especifica qual o tipo da carga atuante na barra e uma variável do tipo *unsigned int* denominada **bar_number** a qual armazena o número da barra onde a carga atua.

A função construtora da classe recebe como parâmetros, na sequência, o número da barra, uma letra para especificar o tipo da carga (as letras válidas são **D** para carga uniformemente distribuida, **E** para carga uniformemente distriguida ao longo do eixo longitudinal da peça, e **T** para carga linearmente distribuida), o valor da carga, o comprimento da barra, e, caso a análise esteja sendo feita no espaço tridimensional, o valor do ângulo entre a carga e o eixo local Y_m (ângulo β da figura 5.9), se for diferente de zero (para ângulo zero o método assume o valor *default* da declaração). Com exceção do número da barra, os outros dados são utilizados para inicializar a variável ponteiro **fptrcarg**.

O método **Chang_Load** requer que sejam passados como parâmetros o novo tipo da carga (se for passada a mesma letra da variável **tip** o efeito será apenas de alteração no valor da carga), o valor da carga, o comprimento da barra onde a carga atua e, se for o caso, o valor do ângulo entre a carga e o eixo local Y_m. Note-se que esses parâmetros estão declarados com valores *default*, significando que se não for fornecido qualquer um deles, o método solicitará que o usuário o faça através do teclado em tempo de execução.

Listagem 5. 22 - Parte do arquivo de cabeçalho da classe **TBar_Load**

```
                                    class TBar_Load
{
 protected :
        TGeneric_Load *fptrcarg ;
        char tip ;
        unsigned int bar_number ;
 public :
        TBar_Load( unsigned int number_bar, char type, float load_value, float bar_size,
                   float angulo=0.0 )          // construtora. OBS.: para a variável angulo
              // somente devera ser atribuido valor caso a carga esteja sendo atribuida a um
              //elemento de barra tridimensional. Caso contrario o metodo ira ignorar esse valor.
        ~TBar_Load( ) { delete fptrcarg ; }
        unsigned int Get_Bar_Number( ) { return(bar_number) ; }  // retorna o numero da
                                                  // barra sujeita a carga (objeto TBar_Load)
        char Get_Tipo ( ) { return (tip) ; }      // retorna qual o tipo da carga
        void Chang_Load( char loadtype='A', float load= 0.0, float barsize=0.0, float ang=0.0 )
              // permite alteracao no tipo e/ou valor da carga. OBS.: para a variável angulo
              // somente devera ser atribuido valor caso a carga esteja sendo atribuida a um
              // elemento de barra tridimensional. Caso contrario o metodo ira ignorar esse
              // valor.
        void Get_Equiv_Load(TVetFloat * fvet_equiv_carg=NULL)      // retorna a carga
                                                                   // nodal equivalente
        void Show ()          { fptrcarg->Show_Carg( ) ; }
        float Mxy_ini ()      { return ( fptrcarg->Get_M1z ( ) ) ; }
        float Mxy_fin ()      { return ( fptrcarg->Get_M2z ( ) ) ; }
        float Myz_ini ()      { return ( fptrcarg->Get_M1x ( ) ) ; }
        float Myz_fin ()      { return ( fptrcarg->Get_M2x ( ) ) ; }
        float Mxz_ini ()      { return ( fptrcarg->Get_M1y ( ) ) ; }
        float Mxz_fin ()      { return ( fptrcarg->Get_M2y ( ) ) ; }
        float Fx_ini ()       { return ( fptrcarg->Get_F1x ( ) ) ; }
        float Fx_fin ()       { return ( fptrcarg->Get_F2x ( ) ) ; }
        float Fy_ini ()       { return ( fptrcarg->Get_F1y ( ) ) ; }
        float Fy_fin ()       { return ( fptrcarg->Get_F2y ( ) ) ; }
        float Fz_ini ()       { return ( fptrcarg->Get_F1z ( ) ) ; }
        float Fz_fin ()       { return ( fptrcarg->Get_F2z ( ) ) ; }
};
```

Através do ponteiro para um objeto da classe **TVetFloat** que lhe é passado como parâmetro, o método **Get_Equiv_Load** retorna o carregamento nodal equivalente para a carga atuante. Neste trabalho implementou-se dentro dessa função uma checagem na ordem do "vetor" passado como parâmetro (através do método **Get_Size**) e, caso o valor obtido seja diferente de 6 ou 12 (ordem que o vetor de carga nodal equivalente deve ter no plano ou no espaço tridimensional, respectivamente), será exibida uma mensagem de erro e a execução do programa será encerrada.

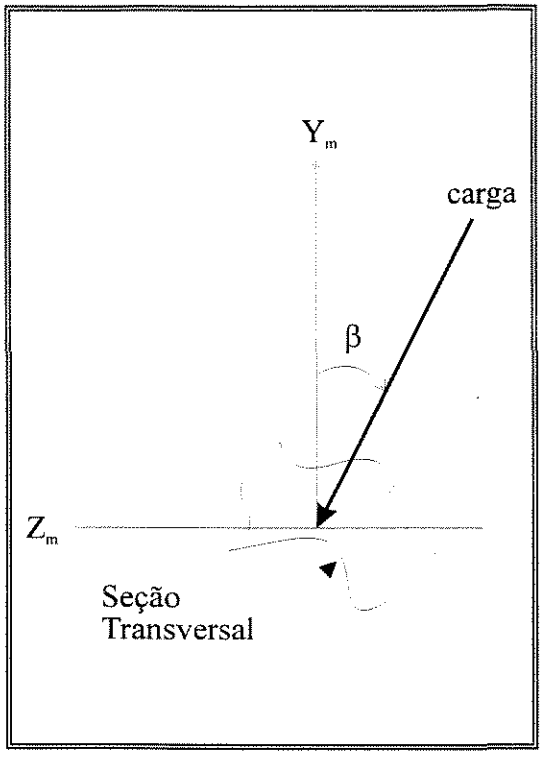

Figura 5.9 - Ângulo de inclinação das cargas na análise tridimensional

Outros métodos implementados retornam, após realizarem os cálculos necessários, os momentos de engastamento perfeito (momentos nodais) e as reações nodais nos nós inicial (estensão ***ini***) e final (extensão ***fin***).

Perceba-se que para cada carga atuante em uma barra haverá um objeto da classe **TBar_Load** e que para um conjunto de cargas atuantes em um único nó existe um objeto da classe **TNode_Load**. A idéia básica do modo como o sistema deve tratar um carregamento completo (todas as cargas que atuam na estrutura) consiste em armazenar esses objetos em listas específicas: uma para armazenar ponteiros que enderecem objetos cargas nas barra (objetos **TBar_Load**) e outra para ponteiros endereçando objetos cargas nos nós (objetos **TNode_Load**). Dessa forma, implementou-se a classe **TLoad2d** (e sua derivada **TLoad3d**) que contém em sua área **protected** duas variáveis ponteiros para objetos da classe **TList** denominadas **flist_node_load** e **flist_bar_load**. Esses ponteiros endereçam, respectivamente, a lista de cargas nos nós e a lista de cargas nas barras. As

considerações sobre os métodos feitas à seguir são pertinentes tanto para a classe **TLoad2d** quanto para a sua classe derivada, com exceção ao método construtor da classe **TLoad3d** que nada tem implementado a não ser a chamada ao método construtor de sua classe raiz.

A função construtora dessa classe requer como parâmetro de entrada um número que indique o tamanho da lista de cargas. Caso esse número não seja fornecido, o método assume o valor *default* da declaração que é definido pela macro **SIZEOF_LISTLOAD** no início do arquivo de cabeçalho. O método destrutor se encarrega de remover todos os objetos armazenados nas listas, utilizar o comando **delete** sobre cada um deles e, quando a lista estiver vazia, utilizar o mesmo comando sobre ela, devolvendo ao sistema a quantidade de memória que estava reservada.

Para atribuir um carregamento à um nó que não possua nenhuma carga utiliza-se o método **Set_Node_Load**. Esse método necessita como parâmetros de entrada o número do nó sobre o qual o carregamento atua e um ponteiro para um objeto **TVetFloat** que endereça o "vetor" onde estão armazenadas as cargas (segundo o esquema já mostrado pela tabela 5.2). Após inicializar um objeto da classe **TNode_Load** o método o "armazena" na lista de cargas nos nós. Caso o número do nó não seja passado na chamada ao método, este solicitará ao usuário que o forneça pelo teclado em tempo de execução. Se houver necessidade de se atribuir carga em uma determinada direção a um nó que já está associado a um carregamento, no qual exista ou não carga na direção desejada, ou se for necessário alterar o valor de uma carga, deve-se utilizar o método **Chang_Node_Load**.

Listagem 5. 23 - Parte do arquivo de cabeçalho da classe **TLoad2d**

```
#define SIZEOF_LISTLOAD  3
                                        class TLoad2d
{
 protected:
          TList * flist_node_load ;
          TList * flist_bar_load ;
 public :
          TLoad2d(unsigned int number_of_load=SIZEOF_LISTLOAD) ;
          virtual ~TLoad2d( ) ;

          virtual void Chang_Node_Load( ) ;          // altera uma carga de um node. O metodo
                    // solicita que o usuario forneca pelo teclado o numero do node, o eixo ou plano
                    // que tera a carga alterada e o valor da carga em tempo de execucao.
          virtual void Remove_Node_Load( ) ;         // remove todas as cargas (objeto TNode_Load)
                    // de um node. Solicita que o usuario fornece o numero do node que tera todas as
                    // cargas removidas. Retira esse objeto TNode_Load da lista de cargas em nodes
                    // (flist_node_load) e deleta-o.
          virtual void Remove_Bar_Load( ) ;          // Solicita o numero da barra que tera todas as
                    // cargas que sobre ela atuam removidas. Retira esses objetos da lista de cargas em
                    // barras (flist_bar_load) e deleta-os.
          virtual void Set_Node_Load( unsigned int node_number=0, TVetFloat * fvetload=NULL) ;
                    // adiciona cargas a um nodo que nao possui nenhum carregamento
          virtual void Set_Bar_Load( unsigned int bar_number, float size_bar, char load_type,
                                        float load,   float angulo=0.0) ;    // adiciona carga em
                    // uma barra. OBS.: para objetos TLoad2d nao deve ser fornecido valor para o
                    // parametro angulo. Se for fornecido valor o metodo ira ignorar. Se for fornecido
                    // o método ira ignorar
          virtual unsigned int Get_Used_List_Node_Load( )
                     { return(flist_node_load->Get_Used( )) ; }          // retorna a quantidade de
                                                                         // nodes com carga
          virtual unsigned int Get_Used_List_Bar_Load( ) { return(flist_bar_load->Get_Used( )) ; }
                                   // retorna a quantidade de cargas em barras
          virtual void Get_All_Node_Load( unsigned int ptr_number, unsigned int & number_node,
                                        TVetFloat * fvet_carg ) ;          // retorna o numero
                    // do node com carga (dentro da variavel number_node) e o vetor de carga
                    // (enderecado por fvet_carg)
          virtual void Get_Equiv_Bar_Load( unsigned int ptr_number, unsigned int & number_bar,
                                        TVetFloat * fvet_carg ) ;// retorna o numero da barra
                    // com carga (dentro da variavel number_bar) e o vetor de carga equivalente nodal
                    // (enderecado por fvet_carg)
} ;
```

As cargas nas barras são atribuidas com a utilização do método **Set_Bar_Load** havendo a necessidade de se efetuar uma chamada ao método para cada carga que atuar na barra. Deve-se fornecer como parâmetros, na sequência, o número da barra, seu comprimento, a letra que indica qual o tipo da carga, valor (máximo no caso de linearmente distribuida) da carga e, se for o caso, o ângulo entre a direção de aplicação da carga e o eixo local Y_m.

O método **Get_Equiv_Bar_Load** calcula o vetor de cargas nodais equivalentes com relação ao sistema de eixos local, retornando-o por meio do ponteiro que lhe é passado como parâmetro (denominado **fvet_carg**), havendo a necessidade de se fornecer a posição na lista de cargas nas barras (variável **flist_bar_load**) da carga que se deseja obter o carregamento equivalente. Outro dado que o método retorna é o número da barra (através do parâmetro **number_bar** passado por endereço) na qual a carga atua, para que possa ser feita a contribuição no vetor de cargas global do resultado da multiplicação do "vetor" de cargas nodais equivalente pela matriz de rotação de eixos da barra em questão. Nessa contribuição deve-se observar as posições das cargas mostradas na tabela 5.2.

Para obtenção do carregamento que atua sobre um nó utiliza-se o método **Get_All_Node_Load**. Neste método o número do nó que está associado à carga aplicada é retornado por intermédio do parâmetro **number_node** passado por endereço e há a necessidade de se fornecer a posição na lista de cargas nos nós (variável **flist_node_load**) de onde será obtido o carregamento. O carregamento será retornado em forma de "vetor" pelo ponteiro passado como parâmetro segundo o esquema da tabela 5.2 e já se apresenta no sitema global de coordenadas.

Com a utilização de um comando **for** que tem como limite o resultado retornado pelo método **Get_Used_List_Bar_Load** percorre-se toda a lista de cargas da barra obtendo de cada um dos objetos ali armazenados o correspondente vetor de carga nodal equivalente, e faz-se, depois da mudança de eixos, as contribuições no vetor de carga total da estrutura. Da mesma forma, com outro comando **for** cujo limite é o valor retornado pelo método **Get_Used_List_Node_Load** percorre-se toda a lista de cargas nos nós, obtendo de cada um dos objetos ali armazenados o correspondente "vetor" de carregamento e faz-se a contribuição dele no vetor de carga global, obtendo-se, assim, o vetor de carga total do modelo.

Deve ser salientado que o procedimento aqui adotado para armazenar as cargas sobre os nós e barras agiliza o processamento, pois percorre-se (através de um comando **for**) uma lista onde os objetos armazenados têm cargas para contribuir na formação do vetor de cargas global; e racionaliza-se a quantidade de memória, pois esta é utilizada apenas com os objetos que realmente dela necessitam para armazenar as cargas que sobre eles atuam.

6 - AS CLASSES QUE GERENCIAM O SISTEMA

Em uma visão abrangente, o problema a ser analisado nada mais é que uma coleção de objetos da classe **TElement** interligados e exercendo uma determinada influência um sobre os outros. É necessário, portanto, uma classe que gerencie a formação dos objetos da classe **TElement**, o momento da sua destruição e os efeitos dessa interligação, permitindo ao usuário executar a análise do problema.

Considerando-se três fatores:

1. Um objeto da classe **TElement** utiliza-se de objetos de outras classes, os quais também necessitam ser criados, destruidos e gerenciados;
2. Alguns desses outros objetos (**TNode**, **TSqMatrix**, **TList**, etc) podem ser utilizados por vários outros tipos de elementos finitos aqui não implementados (elementos triangular CST, triangular LST, quadrático, etc);
3. e, utilizando-se outra vantagem da linguagem C++ chamada herança multipla, a qual permite que uma classe herde dados e métodos de duas ou mais classes,

propõe-se a implementação de várias classes, a maioria derivada de uma classe raiz segundo o esquema da figura 6.1, onde cada classe gerencia um tipo de modelo. Desta forma, proveita-se ao máximo as rotinas de gravação, leitura e criação de objetos aqui desenvolvidas, e evita-se, se possível, reescrevê-las numa futura ampliação. Note-se que a intenção não é que sejam criados objetos dessas classes, muito embora isso possa ser feito, mas sim que o código seja sub-dividido em blocos onde cada um gerencie algum ou alguns tipos de dados e que esses blocos possam ser "unidos" a outros formando novas opções de análise, sem a necessidade de se reescrever o gerenciamento dos tipos de dados já existentes.

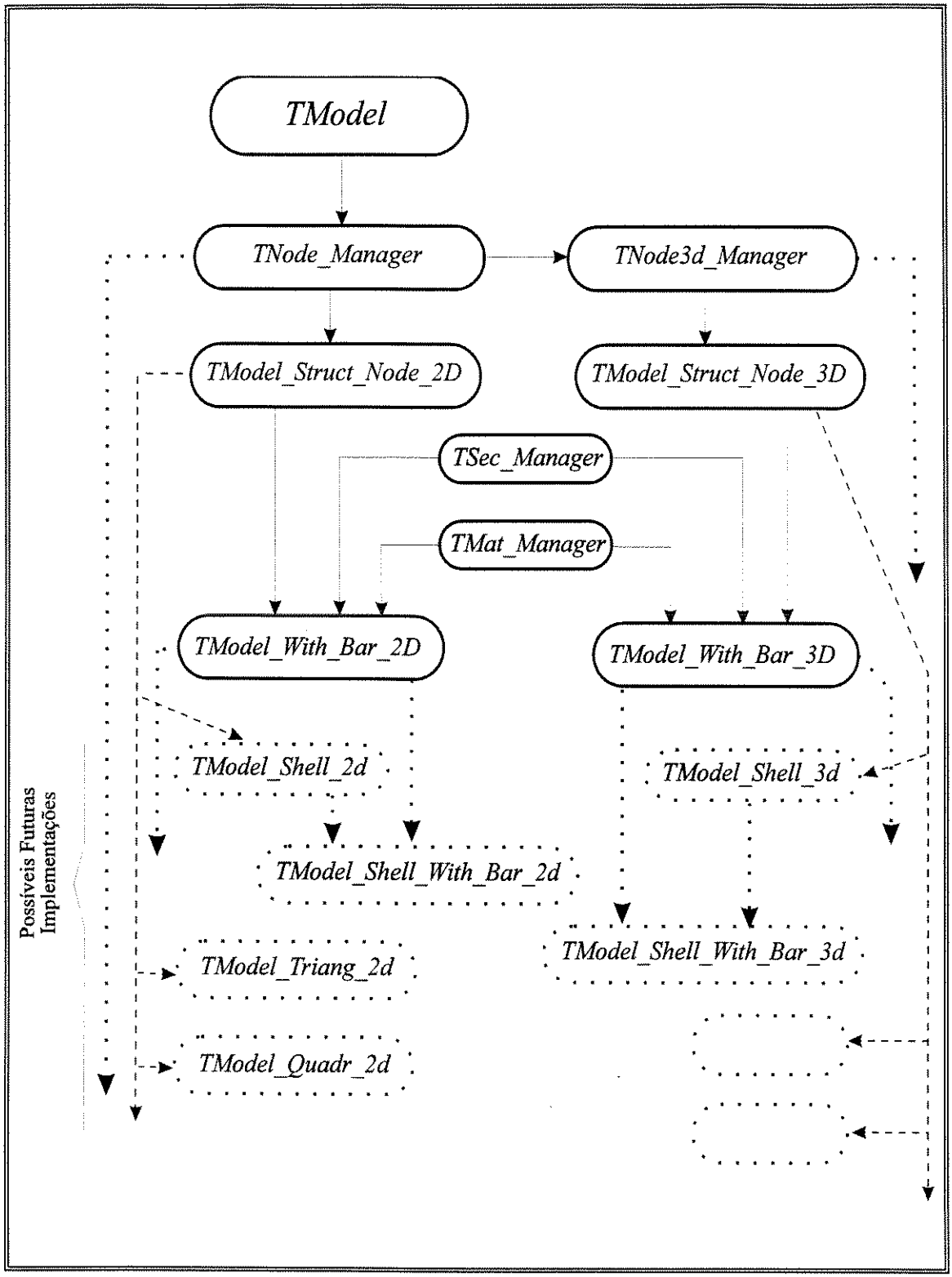

Figura 6.1 - Esquema de Interligação dos Modelos

6.1 - O MODELO (ESTRUTURA)

Com o objetivo de aproveitamento do código, descrito no item anterior, foi desenvolvida a classe **TModel** para representar o modelo que se está analisando. Essa classe contém em sua área **protected** as variáveis **flist_node** e **flist_of_elem**. Essas variáveis endereçam objetos da classe **TList** os quais armazenam ponteiros associados, respectivamente, a objetos da classe **TNode** e objetos da classe **TElement** (ou objetos de classes derivadas destas), que compõem o problema à analisar.

Listagem 6. 1 - Parte do arquivo de cabeçalho da classe **TModel**

```
                                    class TModel
{
 protected :
          TList * flist_node ;
          TList * flist_of_elem ;
          TList * flist_of_model ;
          TVetFloat * fvet_glob ;
          TSqMatrix * fmat_glob ;
          char arq[24] ;

 public :
          TModel( ) ;
          ~TModel( ) ;
          void Get_Model_List( ) { cout<<"este metodo nao esta implantado\n" ; } ;
          virtual void Solve( ) { fmat_glob->Solve(fvet_glob) ; }
          virtual void Read_Data_File( ) { } ;
          virtual void Write_Data_File( ) { } ;
          virtual void Menu( ) { } ;
} ;
```

Compõem, também, a área **protected** desta classe, um vetor de *char* denominado **arq**, uma outra variável ponteiro para objeto da classe **TList** denominada **flist_of_mode**, um ponteiro para um objeto da classe **TVetFloat**, chamado **fvet_glob**, e um ponteiro para um objeto da classe **TSqMatrix** (denominado **fmat_glob)**. Na primeira armazena-se o nome do arquivo no qual o sistema grava os dados do modelo a ser analisado e do qual lê esses valores, os quais serão utilizados

para criar os objetos. O número de caracteres do nome do arquivo deve ser no máximo de oito digitos seguidos ou não de um ponto e três digitos de extensão, isso para atender a exigência do sistema operacional DOS.

Na segunda variável (o ponteiro **flist_of_model**) fornece-se uma opção para a sub-estruturação (não implementada neste trabalho e que possivelmente fará parte de um outro projeto), ou um outro tipo de aplicação do método dos elementos finitos dentro de um modelo mais amplo (o cálculo das características geométricas de uma seção transversal qualquer pode ser considerado como um modelo dentro de um outro, que é a análise da estrutura, por exemplo). Em **fvet_glob** endereça-se o vetor de cargas totais da estrutura (em um outro projeto pode ser um outro vetor de parâmetros conhecidos como dados de fluxo, temperatura, etc), enquanto que em **fmat_glob** armazena-se o endereço da matriz de rigidez da malha de elementos que compõem o problema. Todos esses objetos são inicializados na função construtora com o valor *default* **NULL**.

O método **Solve** desta classe executa a chamada ao método **Solve** associado à matriz (de rigidez global) do problema, passando o "vetor" de valores conhecidos (vetor de cargas nodais neste caso) como parâmetro, para encontrar a resposta ao sistema de equações, que no caso deste trabalho é o vetor de deslocamentos dos pontos nodais da malha de elementos finitos.

Os métodos **Read_Data_File**, **Write_Data_File** e **Menu** são declarados nesta classe com a palavra chave **virtual** e não possuem implementação aqui devendo ser implementados nas classes derivadas desta para que possam executar suas funções, quais sejam: ler os dados do arquivo de dados, preparar (escrever os dados do modelo) esse arquivo e apresentar na tela as opções de interação que o usuário pode ter antes, durante e após a análise do modelo.

6.2 - AS CLASSES QUE GERENCIAM NÓS

6.2.1 - NÓS GEOMÉTRICOS

Derivada da classe **TModel** está a classe **TNode_Manager**, e derivada desta encontra-se a classe **TNode3d_Manager**. Essas classes gerenciam, respectivamente, a formação dos objetos da classe **TNode** e **TNode3D**, a leitura de seus dados, o armazenamento destes dados em arquivos e listas, bem como as opções de alterações em suas configurações.

A função construtora deste método atribui à variável do tipo **float**, denominada **error_weight**, um valor igual ao da macro **ERROR_LIMIT**, definida no início do arquivo de cabeçalho. Essa variável é um parâmetro multiplicador no método que estabelece uma região limite ao redor de um nó (objeto **TNode** ou **TNode3d** já inicializado) na qual, se forem definidas coordenadas de um outro nó, o método responsável por "criá-lo" irá considerar que o nó já existe.

Para se preencher a lista de nós do modelo (endereçada pela variável **flist_node** da classe **TModel**) utiliza-se o método **Fill_List_Node**. Esse método necessita como parâmetro de entrada de um objeto da classe **ifstream** o qual deve ser passado por referência e deve endereçar o arquivo de dados, de onde, primeiramente, o método faz a leitura do número de nós que compõem o modelo. Na sequência, para cada um dos nós da estrutura, lê as coordenadas do nó, faz uma chamada ao método **Test_Node** para verficar se um nó com essas coordenadas (ou próximas) já existe armazenado na lista de nós e, caso não exista, inicializa um objeto **TNode** (ou **TNode3d**) e insere-o no final da lista (**flist_node**).

Listagem 6. 2 - Parte do arquivo de cabeçalho da classe **TNode_Manager**

```
#define ERROR_LIMIT 0.0001
#define EXISTE 0
#define NAOEXISTE 1

                                class TNode_Manager : public TModel
{
 protected :
        float error_weight ;
        virtual void Box_Contour ( TVetFloat * flimit=NULL, float error_limit=ERROR_LIMIT) ;
        virtual unsigned int Test_Node(TVetFloat * fvetcoord=NULL) ;
 public :
        TNode_Manager( ) : TModel( ) ;
        virtual ~TNode_Manager( ) ;
        virtual void Fill_List_Node(ifstream & arq_in) ;          // le os dados dos nodes do
                // arquivo de entrada (objeto da classe ifstream passado por referência). Faz
                // chamada ao metodo Test_Node(TVetFloat *) para cada conjuto de
                // dados de um node lido. Se o node nao existir armazenado, cria-o e armazena-o.
        virtual void Write_Nodes_To_Data_File(ofstream & arq_out) ;     // grava os dados dos
                                                        // node geometricos no arquivo de dados.
        virtual void Create_Node (TVetFloat * fcoord) ;          // cria objetos nodes e
                        // armazena-os na lista de nodes (flist_node). Faz verificacao se o objeto ja
                        // existe. Caso exista, nao cria.
        virtual void Create_Node ( ) ;      // cria objetos nodes e armazena-os na lista de nodes
                        // (flist_node). Faz verificacao se o objeto ja existe. Caso exista, nao cria.
        virtual void Chang_Coord(unsigned int node_number=0) ; // altera uma das coordenadas
                // do node. O metodo solicitara ao usuario que forneca o eixo onde ser alterada a
                // coordenada e, se nao for passado como parametro, o numero do node.
        virtual void Menu_Node( ) ;       // exibe o menu de opcoes para se trabalhar com nodes
        void Remove_One_Node(unsigned int num=0) ;    // remove um node da lista. Se nao for
                        // fornecido o numero do node o metodo solicitara que o usuario o faca
                        // pelo teclado em tempo de execucao
        void Chang_Error_Box_Contour ( ) ;      // altera o valor da variavel error_weight. Essa
                                                // alteracao somente pode ser feita pelo teclado.
        void Show_Node(unsigned int node_number=0) ;  // exibe os dados de um node na tela.
} ;
```

Test_Node é um método cujo acesso é exclusivo aos métodos da classe ou dos métodos de classes dela derivadas (está declarado na área **protected**) e tem como função verificar se o nó já se encontra armazenado na lista. Para tanto, necessita que seja passado como parâmetro um ponteiro que endereçe o "vetor" que contém as coordenadas do nó que será testado. Caso a verificação apresente resultado positivo o método retorna o valor da macro **EXISTE** definida no início do arquivo de cabeçalho, e caso contrário retorna o valor da macro **NAOEXISTE**.

No código do método **Test_Node** é feita uma chamada ao método **Box_Contour**, que, ao percorrer toda a lista de nós, obtém os valores máximos e mínimos das coordenadas. Com esses valores e a variável **error_weight**, calcula as distâncias, em cada um dos eixos, da região em torno de um nó onde um outro não conseguirá ser definido, pois este será confundido com o já existente. Esse método tem como parâmetro de entrada um ponteiro para um objeto **TVetFloat** demonimado **flimit** através do qual retorna o "vetor" que contém o comprimento dos lados da região mensionada. Esse método também é de acesso restrito aos métodos da classe.

O método **Chang_Error_Box_Contour** permite alterar (pelo teclado) o valor da variável **error_weight**. Essa variável é utilizada como multiplicador dos valores obtidos nas subtrações dos máximos e mínimos das coordenadas, alterando-se, assim, o tamanho da região mencionada no parágrafo anterior.

Utilizado para modificar a estrutura já armazenada, existem para o método **Create_Node** duas declarações. Na primeira é passado como parâmetro um ponteiro que endereça um "vetor" onde estão contidos os valores das coordenadas do nó a ser criado. Na outra declaração não há passagem de qualquer parâmetro e, neste caso, o método solicitará ao usuário que forneça as coordenadas. Após obter as coordenadas, faz uma chamada ao método **Test_Node** para testar se um objeto **TNode** com esses dados já existe armazenado em **flist_node**. Se não existir armazenado, o método inicializa-o e armazena-o no final da lista, e, se já existir armazenado, nada será executado, passando-se ao próximo passo da análise.

Para escolher entre as opções de comando que trabalham com os nós utiliza-se o método **Menu_Node** que exibe na tela as seguintes opções:

[A]	=	***Adiciona um novo node a lista de nodes do modelo.***
[C]	=	***Altera as coordenadas de um node.***
[E]	=	***Exibe as coordenadas de um node.***
[M]	=	***Altera limite de erro no calculo de Box-Contour.***
[R]	=	***Remove um node da lista de nodes do modelo.***
[S]	=	***Sair.***

De acordo com a escolha do usuário, o método correspondente será executado.

Para a classe **TNode3d_Manager**, os métodos declarados como virtuais são reescritos de mode a se adequarem aos objetos **TNode3d** que essa classe gerencia. Quanto à variável declarada na área **protected** e aos outros métodos, são herdados sem que haja a necessidade de qualquer alteração.

6.2.2 - NÓS ESTRUTURAIS

Nós estruturais são aqui entendidos como nós geométricos aos quais são imposto deslocamentos definidos. Para reger tais nós no espaço bidimensional, deriva-se de **TNode_Manager** a classe **TModel_Struct_Node_2D**, e para nós no espaço tridimensional implementou-se a classe **TModel_Struct_Node_3D** derivada da classe **TNode3d_Manager**.

Dois ponteiros para objetos da classe **TList**, um denominado **flist_restricao**, que endereça uma lista contendo endereços de objetos da classe **TRestricao**, e outro chamado **flist_recalque**, endereçando uma lista que contém endereços de objetos da classe **TRecalque**, fazem parte da área **protected** das classes **TModel_Struct_Node_2D** (ver arquivo de cabeçalho à seguir) e **TModel_Struct_Node_3D**.

Com a finalidade de gerenciar melhor a quantidade de memória utilizada em uma análise, esses ponteiros são inicializados com **NULL** no método construtor da classe recebendo, posteriormente, endereços de objetos aos quais correspondem. Este procedimento se deve ao fato de que em um modelo com nós estruturais, pode acontecer de nenhum dos nós possuir recalque e poucos possuirem restrições. Neste caso, se uma parte da memória foi reservada para guardar uma lista, ela estaria sendo desperdiçada.

Assim, a inicialização dos ponteiros **flist_restricao** e **flist_recalque**, existentes na área **protected** da classe, se fará apenas quando for realmente necessário atribuir restrição a deslocamento e atribuir recalque, respectivamente. Essas atribuições devem ser feitas com a utilização dos métodos **Set_Restricao**, para restrições, e **Set_Recalque**, para recalques. Estes métodos inicializam os ponteiros **flist_restricao** e **flist_recalque**, passando para o construtor da classe **TList** o valor da macro **SIZEOF_LIST** definida no início do arquivo de cabeçalho.

Existem duas declarações para o método **Set_Restricao**. Em uma há a necessidade de se fornecer como parâmetro de entrada um objeto da classe **ifstream** que é passado por referência. Esse parâmetro deve endereçar o arquivo de dados de onde o método executará a leitura dos valores necessários podendo, assim, atribuir restrições a um ou mais nós, sendo esta declaração do método indicada no início da análise, quando se está lendo os dados da estrutura e preenchendo as variáveis do ambiente. A outra declaração para o método não necessita de qualquer parâmetro de entrada e sua função executará a atribuição de restrição a apenas um nó, sendo esta feita através do teclado (o número do nó e os dados da restrição serão solicitados ao usuário). A utilização desta declaração é indicada para alterar a estrutura já armazenada..

Já existindo pelo menos uma restrição vinculada a um nó (em qualquer direção), o método **Set_Restricao** não será executado para esse nó, devendo-se utilizar o método **Chang_Restricao**. Esse método necessita como parâmetros de entrada do eixo ou plano (um dado do tipo **enum AXIS**, já comentado) que terá possibilidade de deslocamento bloqueada ou liberta e do número do nó. Se esses dados não forem passado na chamada ao método, este solicitará que o usuário o faça pelo teclado em tempo de execução. Caso o nó não esteja vinculado a restrição em qualquer direção, na utilização de **Chang_Restricao** nada será executado, devendo ser chamado o método **Set_Restricao**.

Procedimento idêntico ocorre com os métodos **Set_Recalque** e **Chang_Recalque**.

O método **Set_Recalque** atribui recalques a um ou mais nós através de arquivo ou teclado. A atribuição através de arquivo é feita utilizando-se a primeira declaração para o método que aparece no arquivo de cabeçalho, a qual necessita de um objeto da classe **ifstream** que endereça o arquivo de dados de onde o método fará a leitura dos valores necessários para se atribuir um recalque, enquanto a opção de atribuição de recalque pelo teclado deve utilizar a segunda declaração para o método sendo que essa atribuição ocorrerá em apenas um nó por chamada. Esse método (qualquer que seja a declaração) atribui restrição ao deslocamento no eixo ou plano ao qual for atribuido o recalque.

Listagem 6. 3 - Parte do arquivo de cabeçalho da classe **TModel_Struct_Node_2D**

```
#define SIZEOF_LIST 5
#define LIMIT_RECALQ 0.0001

                    class  TModel_Struct_Node_2D : public TNode_Manager
{
 protected :
        TList * flist_restricao ;
        TList * flist_recalque ;
        virtual void Compare_And_Chang_Restricao( TVetFloat * frecalq,
                                                  unsigned int number_of_node ) ;
 public :
        TModel_Struct_Node_2D( ) : TNode_Manager( ) { } ;
        virtual ~TModel_Struct_Node_2D( ) ;
        virtual void Set_Cond_Contour( ) ;
        virtual void Menu_Node ( ) ;
                                // METODOS QUE ATUAM SOBRE AS RESTRICOES
        virtual void Set_Restricao(ifstream & arq_in) ;
        virtual void Set_Restricao( ) ;
        virtual void Chang_Restricao( AXIS axis=nihil, unsigned int node_number=0) ;
                // altera a condicao de deslocabilidade de um node atraves do teclado ou uma
                // funcao. Se esta liberto, restringe, se esta restrito, libera. Qualquer parametro que
                // for assumido com o valor default sera solicitado durante a execucao.
        virtual void Write_Struct_Nodes_Restric_To_Data_File  (ofstream & arq_out) ;
                                // METODOS QUE ATUAM SOBRE OS RECALQUES
        virtual void Set_Recalque(ifstream & arq_in) ;
        virtual void Set_Recalque( ) ;
        virtual void Chang_Recalque( AXIS axis=nihil, unsigned int node_number=0,
                                     float new_recalque=0.0 ) ;
                // altera a condicao de recalque de um node atraves do teclado ou uma funcao.
                // Qualquer parametro que for assumido com o valor default sera solicitado
                // durante a execucao.
        virtual void Write_Struct_Nodes_Recalq_To_Data_File  (ofstream & arq_out) ;
} ;
```

Chang_Recalque é um método que permite a alteração de um recalque que já esteja vinculado a um nó. Os parâmetros que devem ser fornecidos são o eixo ou plano onde o recalque existe (um dado do tipo **enum AXIS**), o número do nó associado ao recalque e o novo valor para o recalque. Caso esses valores não sejam fornecidos quando da chamada ao método, serão solicitados ao usuário e, se este fornecer um valor igual a zero para o parâmetro **new_recalque** (ou se o método assumir o valor *default* da declaração para esse parâmetro), o método interpretará que deve retirar o recalque imposto ao nó. Esse procedimento não irá retirar a restrição ao deslocamento que foi imposta quando da atribuição do recalque. Para retirá-la deve-se utilizar o método **Chang_Restricao**. Caso o nó não esteja vinculado a recalque em

qualquer direção, na utilização de **Chang_Recalque** nada será executado e deve ser chamado o método **Set_Recalque**.

O método **Compare_And_Chang_Restricao** é utilizado para manter restrita a possibilidade de deslocamento quando se atribui um recalque de apoio ou quando ele é retirado (igualao a zero), isto é, o nó perde o recalque mas continua com o deslocamento naquela direção bloqueado.

Para se incluir as condições de contorno do problema estrutural na matriz de rigidez global (restrições) e no vetor de deslocamentos (recalques) é utilizado o método **Set_Cond_Contour** que somente considerará os recalques que tenham valor absoluto maior que o valor representado pela macro **LIMIT_RECALQ** definida no início do arquivo de cabeçalho

Nota-se que a função **Menu_Node** é sobre-escrita. Isso é feito para permitir a exibição, em uma única tela, de todas as opções existentes que trabalham com nós estruturais, inclusive as utilizadas com nós geométricos, como mudanças de coordenadas por exemplo. Assim, ao ser executada essa função associada a um objeto desta classe ou de outra dela derivada, será exibido o seguinte *menu* de opções:

[A]	=	***Adiciona um novo node a lista de nodes do modelo.***
[C]	=	***Altera as coordenadas de um node.***
[E]	=	***Exibe as coordenadas de um node.***
[F]	=	***Atribui restricoes ao deslocamento em um node.***
[G]	=	***Altera as restricoes ao deslocamento de um node.***
[J]	=	***Atribui recalques de apoio a um node.***
[K]	=	***Altera os recalques de apoio de um node.***
[M]	=	***Altera limite de erro no calculo de Box-Contour.***
[R]	=	***Remove um node da lista de nodes do modelo.***
[S]	=	***Sair.***

A classe **TModel_Struct_Node_3D** é derivada da classe **TNode3d_Manager** e contém as mesmas declarações e variáveis que as existentes na classe **TModel_Struct_Node_2D**, porém com as implementações feitas para gerenciarem objetos da classe TNode3D associados com as restrições e recalques impostos.

Listagem 6. 4 - Esquema de parte do arquivo de cabeçalho da classe **TModel_Struct_Node_3D**

```
class TModel_Struct_Node_3D :  public TNode3d_Manager

{
        // declaracao dos metodos da classe TModel_Struct_Node_3D que são as mesmas de
        // TModel_Struct_Node_2D porem escritas para atuarem com objetos TNode3D.
}
```

6.3 - GERENCIANDO MATERIAIS E SEÇÕES

Considerando-se que neste projeto apenas aos elementos de barra deve ser atribuida seção transversal e um determinado tipo de material (respondem de forma específica a uma solicitação), apenas os problemas que tenham em sua composição esses elementos devem conter um gerenciador para os objetos da classe **TSecao** e **TMaterial**. Assim sendo, não há necessidade de derivar uma classe da classe **TModel** para gerenciar a formação de uma lista de seções e outra para uma lista de materiais, tendo-se optado pela criação de duas classes independentes, denominadas respectivamente **TSec_Manager** e **TMat_Manager**, da qual os modelos compostos por elementos de barra herdarão os dados e métodos, como mostra o esquema da figura 6.1.

TMat_Manager contém em sua área **protected** uma variável da classe **TList** denominada **fmat_list**, inicializada no método construtor com a passagem à função construtora da classe do valor representado pela macro

SIZEOF_MATLIST, (definida no início do arquivo de cabeçalho da classe transcrito à seguir), conforme a sintaxe

fmat_list = new TList (SIZEOF_MATLIST)

Nessa variável devem ser armazenados os ponteiros para os objetos que representam os tipos de materiais que compõem o modelo (objetos da classe **TMaterial**). A intensão com esse procedimento é a de se atribuir à variável **fmaterial** da classe **TBarra** o endereço armazenado por um dos ponteiros dessa lista. Dessa forma, não é armazenado um tipo de material para cada elemento **TBarra** que seja desse material, mas armazena-se na memória os dados de um tipo de material e o endereço "desse material" é atribuido a todos os elementos **TBarra** que sejam desse tipo, economizando-se a memória do sistema.

Para se preencher a lista de materiais pode-se utilizar o método **Fill_Mat_List** ou o método **Add_One_Mat**. Ambos checam se o ponteiro **fmat_list** é igual à **NULL**. Se for, inicializam-no da mesma forma que o método construtor. Após a esse procedimento, inicializam um objeto **TMaterial** e armazenam-o na lista. Para obter os dados do objeto **TMaterial**, o método **Fill_Mat_List** necessita de um objeto da classe **ifstream**, o qual deve ser passado por referência e está associado ao arquivo de dados de onde será feita a leitura dos valores necessários a criação dos tipos de materiais (objetos **TMaterial**). Já o método **Add_One_Mat** solicita ao usuário que forneça pelo teclado e em tempo de execução esses dados. Após a inicialização do objeto **TMaterial**, este terá seu endereço armazenado na lista **fmat_list**.

O método **Remove_One _Mat** remove um tipo de material da lista de materiais (executa um comado **delete** sobre o objeto **TMaterial** após removê-lo da lista, devolvendo ao sistema a memória que ele ocupava). Deve-se fornecer o número da posição na lista onde o endereço do tipo de material à ser removido está armazenado (caso não seja fornecido na chamada, o método solicitará que o usuário o faça).

Semelhante aos outros métodos que fazem gravações no arquivo de dados, o método **Write_Material_To_Data_File** necessita como parâmetro de entrada de um objeto da classe **ofstream**, o qual deve ser passado por referência e estar associado ao arquivo onde os dados serão gravados.

Listagem 6. 5 - Parte do arquivo de cabeçalho da classe **TMat_Manager**

```
#define SIZEOF_MATLIST 2

                                    class TMat_Manager
{
 protected :
        TList * fmat_list ;
 public :
        TMat_Manager ( ) ;
        ~TMat_Manager ( ) ;
        virtual void Fill_Mat_List (ifstream & arq_in) ;    // le os dados do arquivo de dados,
                // inicializa um objeto da classe TMaterial com esse dados e armazena-o na lista.
        virtual void Add_One_Mat ( ) ;              // adiciona um tipo de material na lsita.
        virtual void Write_Material_To_Data_File(ofstream & arq_out) ;   // prepara o arquivo
                                                                         // de dados.
        void Remove_One_Mat (unsigned int mat_number=0) ;                // remove um material
                // da lista de materiais. Deve-se fornecer a posicao na lista onde o endereco do
                // material esta armazenado.
        void Menu_Material() ;              // exibe na tela o menu de opcoes.
} ;
```

Para ter acesso às opções de trabalho com os tipos de materiais deve-se utilizar o método **Menu_Material** que exibe a seguinte tela:

```
Escolha :
[ A ] -> Adiciona um novo material.
[ R ] -> Remove um material.
[ S ] -> Sair.
```

A classe **TSec_Manager** contém em sua área **protected** duas variáveis do tipo ponteiro para objetos da classe **TList** denominadas **flistsecao** e **fgruplist** (inicializadas na função construtora com o valor representado pela macro **SIZEOF_LISTSECAO**). Tais ponteiros devem endereçar objetos que representem, respectivamente, uma lista de seções transversais e uma lista de grupos de seções. Nestas devem ser armazenados os ponteiros para os objetos da classe **TSecao** (seção

transversal para o elemento de barra) e os ponteiros para os objetos da classe **TGrup_Secao** (grupo de seção), respectivamente.

Existe nessa área, também, um método denominado **Test_Secao** que procura se uma determinada seção transversal já existe armazenada na lista de seções (**flistsecao**) e, caso não exista, inicializa um objeto da classe **TSecao** e armazena-o no final da lista. É um método restrito às funções da classe e, neste projeto, é utilizado pelos métodos **Fill_List_Secao** e **Add_One_Secao** para que não haja uma mesma seção transversal armazenada em dobro, ocupando memória do sistema.

O método **Fill_List_Secao** e **Add_One_Secao** têm implementação e funcionamento semelhantes aos métodos **Fill_Mat_List** e **Add_One_Mat** da classe **TMat_Manager**. Assim, **Fill_List_Secao** recebe por referência o objeto da classe **ifstream** associado ao arquivo de dados, lê o número de seções transversais existentes na estrutura e, através de uma instrução **for,** faz as leituras dos dados das seções e chama o método **Test_Secao**, o qual preenche a lista de seções verificando se a seção já existe, criando o objeto **TSecao** caso ela não exista e armazenando-o na lista. **Add_One_Secao** adiciona uma seção com os dados fornecidos pelo usuário através do teclado (não necessita de parâmetros de entrada, tendo, portanto, sua utilização recomendada para se alterar o modelo após este já ter sido criado). Ambos inicializam o ponteiro **flistsecao** caso a comparação deste com **NULL** resulte verdadeira.

Somente após ter-se preenchido a lista de seções, pode-se utilizar os métodos **Fill_Grup_List** e **Add_One_Grup**. O primeiro necessita como parâmetro de entrada de um objeto da classe **ifstream**, o qual deve ser passado por referência e deve estar associado ao arquivo de dados de onde o método lê a quantidade de grupos de seções existentes no modelo. Então, dentro de uma instrução **for**, faz a leitura do número da posição na lista de seções que a seção (leia-se endereço

de um objeto **TSecao**) relativa ao grupo ocupa, obtém o endereço do objeto **TSecao** através do comando

*(TSecao *) flistsecao->Get_Data(numero da posição lido no arquivo)*,

inicializa o objeto **TGrup_Secao** com esse ponteiro e armazena-o na lista. No método **Add_One_Grup** o procedimento é semelhante, havendo a possibilidade de se fornecer como parâmetro de entrada na chamada ao método o número da posição na lista de seções ocupado pela seção (novamente leia-se endereço do objeto **TSecao)** relativa ao grupo, ou aguardar que o método o solicite (procedimento que ocorre quando nenhum valor é passado como parâmetro de entrada e o valor *default* é adotado).

Listagem 6. 6 - Parte do arquivo de cabeçalho da classe **TSec_Manager**

```
#define SIZEOF_LISTSECAO 5
#define SEC_ERROR 0.0001

                                        class TSec_Manager
{
 protected :
        TList * flistsecao ;
        TList * fgruplist ;
        void Test_Secao (sectransv sectr) ;          //struct definida no arquivo my_struc.h
 public :
        TSec_Manager ( ) ;
        ~TSec_Manager ( ) ;
        void Fill_List_Secao (ifstream & arq_in) ;
        virtual void Add_One_Secao ( ) ;
        void Remove_One_Secao (unsigned int sec_number=0) ;
        virtual void Fill_Grup_List (ifstream & arq_in) ;
        void Chang_One_Grup( unsigned int new_sec_number=0, unsigned int grup_number=0 ) ;
        void Add_One_Grup (unsigned int new_grup_sec_number=0) ;
        virtual void Write_Secoes_To_Data_File(ofstream & arq_out) ;
        TList * Get_Grup_List ( ) { return (fgruplist) ; }
        void Menu_Secoes( ) ;
} ;
```

Para se remover uma seção da lista de seções deve-se utilizar o método **Remove_One_Secao**. Na chamada a esse método deve ser fornecido o número da posição na lista em que se encontra o endereço da seção (caso esse número não seja fornecido na chamada, o método solicitará que o usuário o faça pelo teclado em tempo de execução). Durante a execução da função é feita uma verificação na lista de grupos. Àqueles que estiverem vinculados à seção que se pretende remover, será solicitado que o usuário forneça o número da posição de uma outra seção para ser-lhes atribuida (ou seja, o método redireciona a seção para a qual os ponteiros dos grupos - variável **fsecao** da classe **TGrup_Secao** - apontam).

6.4 - GERENCIANDO AS CARGAS DO MODELO

Para gerenciar as cargas atuantes no modelo foram implementadas as classes **TLoad2d_Manager** (utilizada em análises bidimensionais) e a classe **TLoad3d_Manager** (para análises tridimensionais) que, semelhantemente às classe **TSec_Manager** e **TMat_Manager**, não são derivadas de **TModel**, porém, diferentemente de todas as outras classes até aqui detalhadas neste capítulo, foram desenvolvidas para que sejam criados objetos (em verdade, um único) de seu tipo (por esse motivo não constam do esquema da figura 6.1). Esse objeto fará parte das variáveis das classes que gerenciam objetos do tipo **TBarra** ou derivados.

Ambas as classes (**TLoad2d_Manager** e **TLoad3d_Manager**) contém declarada em suas respectivas áreas **protected** uma variável tipo ponteiro para objeto **TList** (denominada **flist_load**) incializada no construtor da classe. Essa variável representa uma lista que deve ser preenchida com endereços de objetos **TLoad2d** ou **TLoad3d,** respectivamente para a classe **TLoad2d_Manager** e a classe **TLoad3d_Manager**.

Cada um desses objetos (**TLoad2d** ou **TLoad3d**) contém uma lista de cargas em nós e uma lista de cargas em barras. O efeito desejado com essa forma de armazenamento é criar uma lista de tipos de carregamentos endereçada por **flist_load**, isto é, o primeiro endereço armazenado nessa variável pode representar, por exemplo, o carregamento permanente, contendo ou não cargas em nós e barras, o segundo, representaria um carregamento acidental (também contendo ou não cargas em nós e barras), o terceiro um outro carregamento acidental, e assim por diante.

Deste modo, a solução do problema é executada resolvendo-se vários sistemas de equações, onde para cada sistema o vetor de cargas totais é devido a um tipo de carregamento. Obtém-se, desta forma, vários vetores de deslocamentos, um para cada tipo de carregamento, que podem ser combinados (somados) de acordo com a intenção do usuário. Aplica-se, assim, a superposição de efeitos, pois este procedimento (combinação dos deslocamentos) representa a atuação simultânea na estrutura dos tipos de carregamentos associados aos deslocamentos combinados.

Cada um dos vetores de deslocamentos é gravado em um arquivo cujo nome é solicitado ao usuário. Para armazenar esse nome, é criado um objeto da classe **TName_Arq** que contém em sua área **protected** uma variável do tipo vetor de *char* com espaço para 20 caracteres. Essa variável armazena o nome do arquivo que contém um dos vetores de deslocamentode modo a poder ser recuperado quando for feita a combinação entre os diversos tipos de carregamentos. Todos os objetos dessa classe são armazenados na variável **flist_name_arq** que representa uma lista de nomes de arquivos.

As combinações entre os carregamentos são representadas por objetos da classe **TComb_of_Load** (listagem 6.7) e são armazenadas na variável **flist_combinations**. **TComb_of_Load** é inicializada com o número de carregamentos que se combinam (adicionam) entre si, sendo tal número utilizado para criar uma lista (variável **flist_of_load_weight** contida na área **protected** da classe) na qual serão armazenados endereços de objetos da classe **TLoad_With_Weight** (listagem 6.8).

Esta última classe é utilizada para manter um tipo de carregamento associado ao seu peso.

Listagem 6. 7 - Parte do arquivo de cabeçalho da classe **TComb_of_Load**

```
                                    class TComb_of_Load
{
 protected :
        TList * flist_of_load_weight ;
 public :
        TComb_of_Load( unsigned int number_of_type_load=0 ) ; // Construtora. Necessita do
                                    // numero de carregamentos que participam da combinacao.
        ~TComb_of_Load( ) ;                         // Destrutora.
        void Add_One_Type_Load(unsigned int type_number=0, float weight=1) ;   // Adiciona
                // um tipo de carregamento a uma combinacao ja existente. Deve-se fornecer o
                // numero do tipo de carregamento e seu peso. Caso o peso nao seja fornecido, o
                // metodo assume o valor 1.
        void Chang_Weight( unsigned int type_number=0, float new_weight=1 ) ;   // Altera o
                // valor do peso de um tipo de carregamento. Deve-se fornecer o numero do tipo de
                // carregamento e o novo valor para o peso.
        float Get_Type_Load_Weight (unsigned int type_number=0) ;       // retorna o valor do
                // peso para o tipo de carregamento passado como parametro.
        void Get_Number_And_Weight( unsigned int position, unsigned int & number_load,
                                    float & weight) ;          // Para a posicao na lista de
                // carregamentos (variavel flist_of_load_weight) fornecida, retorna o numero do
                // tipo de carregamento armazenado e seu peso.
        unsigned int Get_Number_Of_Load_At_Comb( )
                { return (flist_of_load_weight->Get_Used()) ; }      // retorna a quantidade de tipos
                                    // de carregamentos participantes da combinacao.
} ;
```

Listagem 6. 8 - Parte do arquivo de cabeçalho da classe **TLoad_With_Weight**

```
class TLoad_With_Weight
{
 protected:
        unsigned int number_of_load_type ;      // armazena o numero da posicao na lista de
                                    // carregamentos do tipo de carregamento associado ao objeto.
        float weight ;
 public:
        TLoad_With_Weight( unsigned int number=0, float peso=1 ) ;      // Construtora.
                // Necessita do numero do tipo de carregamento e seu peso.
        ~TLoad_With_Weight( ) { } ;     // Destrutora.
        void Chang_Weight( float new_weight ) ; // Altera o peso do tipo de carregamento.
        float Get_Weight ( ) { return(weight) ; }               // Retorna o valor do peso associado ao
                                                                // tipo de carregamento.
        unsigned int Get_Number_of_Load_Type( )
                        { return(number_of_load_type) ; }       // retorna o numero da posicao
                // na lista de carregamentos do tipo de carregamento associado ao objeto.
        void Get_Number_And_Weight( unsigned int & load_number, float & load_weight ) ;
                // Retorna o numero da posicao na lista de carregamentos do tipo de carregamento
                // associado ao objeto e seu peso.
} ;
```

Voltando-se à classe **TLoad2d_Manager**, nota-se pela listagem 6.9 que vários dos métodos já se encontram comentados logo após a sua declaração.

O método **Get_Node_Vet_Load** retorna o número do nó através do parâmetro **node_number** passado por referência e, endereçado pelo ponteiro **fvet_contrib_load**, o vetor de cargas que atuam nesse nó para que seja feita a contribuição desses valores no vetor de cargas global. Para tanto, nessecita como parâmetros de entrada o número do tipo de carregamento e o número da posição que a carga ocupa na lista de cargas atuantes sobre nós do objeto **TLoad2d** (ou **TLoad3d**, se for o caso) associado a esse carregamento.

Efeito idêntico ocorre com a utilização do método **Get_Bar_Equiv_Load**. Passando-se como parâmetros de entrada o número do tipo de carregamento e o número da posição que a carga ocupa na lista de cargas atuantes em barras do objeto **TLoad2d** (ou **TLoad3d**, se for o caso) associado a esse carregamento, o método retorna o número da barra, através do parâmetro **bar_number** passado por referência, e o vetor de cargas equivalentes, através do ponteiro denominado **fcontrib_load**. Esses dados serão utilizados na montagem do vetor de cargas globais.

Note-se que para esse método (**Get_Bar_Equiv_Load**) existe uma segunda declaração na qual devem ser fornecidos como parâmetros de entrada o número do tipo de carregamento e o número da barra da qual se deseja obter o vetor de cargas equivalentes (retornado pelo ponteiro **fvetcarg**).

Listagem 6. 9 - Parte do arquivo de cabeçalho da classe **TLoad2d_Manager**

```
#define CARREGAMENTOS_2D 2
                                        class TLoad2d_Manager
{
 protected :
        TList * flist_load ;
        TList * flist_combinations ;
        TList * flist_name_arq ;
 public :
        TLoad2d_Manager( ) ;
        virtual ~TLoad2d_Manager( ) ;
        virtual void Create_One_Load_Type( ) ;          // inicializa um tipo de carregamento
                                                        // (objeto TLoad2d).
        virtual void Remove_All_Bar_Load( ) ;           // para um certo tipo de carregamento,
              // remove todas as cargas que atuam em uma determinada barra. Os dados serao
              // solicitados.
        virtual void Remove_All_Node_Load( ) ;  // para um certo tipo de carregamento,
              // remove todas as cargas que atuam em um determinado node. Os dados serao
              // solicitados.
        virtual void Chang_One_Node_Load( ) ;           // para um certo tipo de carregamento,
              // altera uma das carga que atuam em um determinado node. Os dados serao
              // solicitados.
        virtual void Attrib_Load_To_Node( ) ;           // para um certo tipo de carregamento,
              // atribui uma carga a um determinado node que ainda nao tenha nenhuma carga
              // aplicada. Os dados serao solicitados.
        virtual void Attrib_Load_To_Node( unsigned int type_load, unsigned int node_number,
                                          TVetFloat * fvetload ) ;
        virtual void Attrib_Load_To_Bar( unsigned int type_load, unsigned int bar_number,
                                         float bar_size, char load_type, float load_value) ;
        virtual void Attrib_Load_To_Bar( unsigned int bar_number, float bar_size ) ;
              // atribui carga a uma barra. O metodo solicitara o numero do tipo de
              // carregamento, o tipo de carga e seus dados.
        unsigned int Get_Number_Of_Type_Load( ) { return(flist_load->Get_Used( )) ; }
              // retorna o numero de tipos de carregamentos
        unsigned int Get_Number_Of_Node_With_Load_On_Type(unsigned int type_number);
              // O parametro de entrada indica a posicao ocupada pelo endereco do
              // carregamento  na lista de tipos de carregamento. Para o tipo de carregamento
              // passado como parametro retorna o numero de nodes que tem carga atuando
              // sobre eles.
        unsigned int Get_Number_Of_Bar_With_Load_On_Type (unsigned int type_number);
              // O parametro de entrada indica a posicao ocupada pelo endereco do
              // carregamento  na lista de tipos de carregamento. Para o tipo de
              // carregamento passado como parametro retorna o numero de cargas que
              // atuam em barras.
        virtual void Get_Node_Vet_Load( unsigned int type_load, unsigned int load_position,
                                        unsigned int & node_number, TVetFloat * fvet_contrib_load ) ;
        virtual void Get_Bar_Equiv_Load( unsigned int type_load, unsigned int bar_position,
                                         unsigned int & bar_number, TVetFloat * fcontrib_load
) ;
```

Listagem 6. 9 - Continuação

```
        virtual void Get_Bar_Equiv_Load( unsigned int type_load, unsigned int bar_number,
                                         TVetFloat * fvetcarg ) ;
        virtual void Set_List_Comb(unsigned int number_of_comb=0) ;      // inicializa a variavel
                // flist_combinations com a quantidade de combinacoes entre carregamentos. Essa
                // quantidade deve ser fornecida atraves do parametro number_of_comb passado
                // na chamada do metodo.
        virtual void Set_Comb_Numb_Type_Load( unsigned int number_of_load=0) ;        //
                // adiciona um tipo de combinacao a lista de combinacoes (variavel
                // flist_combinations). O numero de tipos de carregamentos que fazem parte do tipo
                // de combinacao adicionado deve ser fornecido na chamada do metodo.
        virtual void Add_Load_On_Comb( unsigned int comb_number=0,
                                       unsigned int load_number=0, float weight=0 ) ;
                // Adiciona um tipo de carregamento a uma combinacao de carregamentos ja
                // existente. Deve ser fornecido o numero da posiciao na lista de combinacoes
                // (variavel flist_combinations) ocupada pela combinacao que sera
                // modificada, o numero do tipo de carregamento e seu peso.
        virtual void Get_Arq_In(unsigned int position, ifstream & arq ) ;    // permite, atraves da
                // variavel arq passada para o metodo, o acesso ao arquivo correspondente ao
                // objeto TName_Arq armazenado na posicao position (fornecida na chamada
                // ao metodo) da lista de nomes de arquivos (variavel flist_name_arq).
        virtual void Insert_Name_Arq(TName_Arq * fname_arq) ; // adiciona um nome de
                // arquivo (em verdade um objeto do tipo TName_Arq) na lista de nomes de
                // arquivos (variavel flist_name_arq).
        virtual unsigned int Number_Of_Comb( ) ;          // retorna o numero de combinacoes que
                // devem ser analisadas entre os tipos de carregamentos atuantes.
        virtual unsigned int Get_Number_Of_Load_At_Comb (unsigned int combin_number) ;
                // retorna o numero de tipos de carregamentos que devem ser considerados na
                // combinacao em questao. O numero da combinacao (posicao que ela ocupa
                // na lista de combinacoes - variavel flist_combinations) deve ser fornecido na
                // chamada ao metodo.
        virtual void Get_Number_And_Weight( unsigned int comb, unsigned int position,
                                            unsigned int & load_number, float & weight ) ;
                // retorna na variavel load_number o numero do tipo de carregamento e na
                // variavel weight o peso que deve multiplicar esse tipo de carregamento na
                // composicao da combinacao. Deve-se fornecer, atraves da variavel comb, a
                // posicao na lista de combinacoes daquela que se deseja obter os resultados.
                // Deve-se fornecer também, atraves da variavel position, a localizacao na lista
                // de tipos de carregamentos que compoem a combinacao do tipo de
                // carregamento que se deseja obter os dados acima..
};
```

Para a classe **TLoad3d_Manager** existem as mesmas declarações que as existentes na classe **TLoad2d_Manager**, porém com praticamente todos os métodos reescritos. Por possuir as mesmas declarações, o arquivo de cabeçalho dessa classe é semelhante ao da classe **TLoad2d_Manager**, ocorrendo uma diferença apenas com o método **Attrib_Load_To_Bar** que necessita como parâmetros de entrada do tipo de carregamento, número da barra, comprimento da

barra, tipo da carga atuante, valor da carga e do angulo entre a carga e o eixo Y do elemento (local).

6.5 - GERENCIANDO O MODELO COMPOSTO POR BARRAS

A classe **TModel_With_Bar_2D** contém as variáveis (no caso um ponteiro para objetos da classe **TLoad2d_Manager** denominado **fload_manager**) e os métodos gerenciadores relacionados aos elementos de barra que compõem a lista de elementos (variável **flist_of_elem** herdada da classe **TModel**). Essa classe é o primeiro exemplo de herança multipla neste projeto, pois ela é derivada das classes **TModel_Struct_Node_2D**, **TSec_Manager** e **TMat_Manager**, das quais herda dados e métodos. O ponteiro **fload_manager**, inicializado no método construtor da classe, é o responsável por gerenciar todas as operações que envolvam cargas.

De modo semelhante está declarada a classe **TModel_With_Bar_3D**. Derivada das classes **TModel_Struct_Node_3D**, **TSec_Manager** e **TMat_Manager**, também é um exemplo de herança múltipla, e contém em sua área **protected** um ponteiro para objetos da classe **TLoad3d_Manager**, também denominado **fload_manager**, o qual é inicializado no método construtor da classe e é utilizado para gerenciar as operações que envolvam cargas em elementos de barra tridimensionais. Os métodos aos quais esta classe responde são os mesmos que os atendidos pela classe **TModel_With_Bar_2D**, desse modo, os arquivos de cabeçalhos de ambas são idênticos (exceção feita aos métodos construtor e destrutor, pela razão do nome das classes).

Listagem 6. 10 - Sequência de instruções do método **Read_Data_File**

```
{
        ifstream in_arq ;

        cout<<"Nome do arquivo de dados - maximo (8 digitos)+(.)+(3digitos) = " ;
        gets(data_arq) ;
        cout<<"\n" ;
//abrindo o arquivo de dados
        in_arq.open(data_arq) ;
        if ( in_arq.fail()!=0 )
          error("Impossivel abrir arquivo.",SAIR) ;
        Fill_List_Node(in_arq) ;
        Set_Restricao(in_arq) ;
        Set_Recalque(in_arq) ;
        Fill_List_Secao(in_arq) ;
        Fill_Grup_List(in_arq) ;
        Fill_Mat_List (in_arq) ;
        Read_Bars_From_Data_File(in_arq) ;
        Read_Load_From_Data_File(in_arq) ;
        in_arq.close() ;
}
```

Listagem 6. 11 - Sequência de instruções do método **Write_Data_File**

```
{
        ofstream out_arq ;
        cout<<"Nome do arquivo de dados - maximo (8 digitos)+(.)+(3 digitos) = " ;
        gets(data_arq) ;
        cout<<"\n" ;
//abrindo o arquivo de dados
        out_arq.open(data_arq) ;
        if ( out_arq.fail()!=0 )
          error("Impossivel abrir arquivo.",SAIR) ;
        Write_Nodes_To_Data_File(out_arq) ;
        Write_Struct_Nodes_Restric_To_Data_File(out_arq) ;
        Write_Struct_Nodes_Recalq_To_Data_File(out_arq) ;
        Write_Secoes_To_Data_File(out_arq) ;
        Write_Material_To_Data_File(out_arq) ;
        Write_Bars_To_Data_File(out_arq) ;
        Write_Load_To_Data_File(out_arq) ;
        out_arq.close() ;
}
```

Read_Bars_From_Data_File é um método utilizado para preencher a variável **flist_of_elem**. Necessita, como parâmetro de entrada, de um objeto da classe **ifstream** passado por referência e associado ao arquivo de onde o método fará a leitura dos dados relativos às barras. Para cada uma, obtém das respectivas listas já existentes e preenchidas (nós, cargas, seções, etc) os objetos

Para realizar a tarefa de ler os dados dos diversos objetos a partir de um arquivo, utiliza-se o método **Read_Data_File** cuja implementação é simples e deverá ser seguida num possível aumento da potencialidade deste projeto. Através da sequência de comandos mostrada na listagem 6.10, todos os dados sobre os objetos serão lidos e eles inicializados preenchendo-se as variáveis que representam a estrutura sob análise. Note-se que a sequência de leitura do arquivo deve ser a mesma que a da escrita e que deve ser observada a interligação entre os objetos. Assim, não é possível criar um objeto que represente carga em barra se a barra ainda não foi criada, ou inicializar uma barra sem ter preenchido a lista de nós.

A gravação dos dados da estrutura em um arquivo pode ser feita com a utilização do método **Write_Data_File**. Esse método realiza chamadas aos vários métodos de escrita das diversas classes, como mostra a listagem 6.11. Deve-se ressaltar novamente que a sequência de gravação dos dados proposta neste método ou em qualquer outro que venha a substituí-lo, deve ser a mesma que a da leitura desses dados (método **Read_Data_File**). O Apêndice A traz um roteiro de preparação do arquivo de dados que o método **Read_Data_File** aqui implementado aceita. Qualquer alteração na execução desse roteiro deve ser acompanhada de uma alteração correspondente no método de leitura. A preparação desse arquivo de dados pode ser feita em qualquer editor de texto que edite em formato **ASCI**.

necessários à inicialização de um objeto **TBarra** ou **TBarra3d**. Cria o objeto e armazena-o no final da lista.

O método **Solve** realiza todo o procedimento para se obter a solução do problema. Primeiramente, inicializa as variáveis **fvet_glob** e **fmat_glob** (ambas herdadas da classe **TModel**, porém sem estarem inicializadas). Com uma chamada ao método **Fill_Mat_Glob** (declarado na área **protected**) preenche a variável **fmat_glob** com a matriz de rigidez da estrutura, após o que, grava os dados desta matriz em um arquivo denominado **GLOBMATR.CAL**. Esse procedimento é importante para se agilizar a análise quando existem mais de um vetor de carga, ou seja, vários sistemas de equações à serem resolvidos. Nesse caso não será necessário realizar todos os cálculos novamente para se obter a matriz de rigidez, bastando lê-los do arquivo e armazená-los em **fmat_glob**, pois esta, após a solução do sistema de equações, não mais contém os dados da matriz de rigidez (ver comentário do método **Solve** da classe **TSqMatrix**).

Após gravar a matriz o método verifica qual o número de tipos de carregamentos (permanente, acidental 1, acidental 2, etc) e, dentro de uma instrução **for** que tem esse número como limite, preenche o vetor de cargas globais para um tipo de carregamento, lê do arquivo previamente gravado os dados da matriz de rigidez, resolve o sistema através do comando

fmat_glob->Solve(fvet_glob)

e armazena a solução para cada um dos sistemas de equações (os resultados obtidos que agora estão armazenados na variável **fvet_glob**), em um arquivo cujo nome é solicitado ao usuário. Esse procedimento é feito com uma chamada ao método **Save_Global_Result**.

Listagem 6. 12 - Parte do arquivo de cabeçalho da classe **TModel_With_Bar_2D**

```
                class TModel_With_Bar_2D : public TModel_Struct_Node_2D,
                                           public TSec_Manager, public TMat_Manager
{
 protected :
        TLoad2d_Manager * fload_manager ;
        virtual void Fill_Vet_Glob(unsigned int number_of_type_load) ;      // preenche o vetor
                                                                            // global
        virtual void Fill_Mat_Glob( ) ;                    // preenche a matriz global
        virtual void Save_Global_Result() ;          // grava o vetor global em um arquivo
public :
        TModel_With_Bar_2D ( ) ;
        ~TModel_With_Bar_2D ( ) ;
        void Chang_Bar_Ni( ) ;            // altera o node inicial da barra. O metodo solicitara
                // ao usuario o numero da barra que tera o node inicial substituido e o numero do
                // node que o substituira.
        void Chang_Bar_Nf( ) ;            // altera o node final da barra. O metodo solicitara
                // ao usuario o numero da barra que tera o node final substituido e o numero do
                // node que o substituira.
        void Chang_Bar_Mat( ) ;// altera o material que define o comportamento da barra. O
                // metodo solicitara o numero da barra que tera o tipo de material substituido
                // e o numeo da posicao na lista de material (fmat_list) do novo material.
        void Chang_Bar_Grup_Sec( ) ;              // altera o grupo de secao ao qual a barra
                // pertence. O metodo solicitara o numero da barra que tera o grupo substituido e o
                // numero da posicao na lista de grupos (fgruplist) daquele que ira substitui-lo.
        virtual void Set_One_Bar( ) ;
        virtual void Remove_One_Element ( ) ;
        virtual void Menu_Bar( ) ;        // exibe na tela o menu de opcoes para objetos de barra.
        virtual void Menu_Load( ) ;                // exibe na tela o menu de opcoes objetos carga.
        virtual void Solve( ) ;
        virtual void Save_Element_Result( ) ;     // para o vetor de deslocamentos globais lido de
                // um arquivo, calcula e grava em outro arquivo as solicitacoes emcada elemento
        virtual void Show_Result( ) ;     // exibe na tela os resultados globais ou de um elemento
        virtual void Main_Menu( ) ;
        virtual void Read_Bars_From_Data_File (ifstream & arq_in) ;      // le os dados das
                // barras do arquivo de dados. Necessita de um parametro passado por referencia.
                // Esse parametro deve ser um objeto da classe ifstream associado ao arquivo de
                // onde sera feita a leitura dos dados.
        virtual void Write_Bars_To_Data_File (ofstream & arq_out) ;      // grava os dados das
                // barras no arquivo de dados. Necessita de um parametro passado por referencia.
                // Esse parametro deve ser um objeto da classe ofstream associado ao arquivo onde
                // sera feita a gravacao dos dados.
        virtual void Read_Load_From_Data_File(ifstream & arq_in) ;      // le os dados das
                // cargas do arquivo de dados. Necessita de um parametro passado por referencia.
                // Esse parametro deve ser um objeto da classe ifstream associado ao arquivo de
                // onde sera feita a leitura dos dados.
        virtual void Write_Load_To_Data_File(ofstream & arq_out) ;      // grava os dados das
                // cargas no arquivo de dados. Necessita de um parametro passado por referencia.
                // Esse parametro deve ser um objeto da classe ofstream associado ao arquivo onde
                // sera feita a gravacao dos dados.
        virtual void Read_Data_File( ) ;          // le todo o arquivo de dados.
        virtual void Write_Data_File( ) ;         // escreve todo o arquivo de dados.
} ;
```

O método **Save_Element_Result** calcula e grava, para cada barra, seu número e os valores dos esforços nos nós inicial e final. Para tanto, solicita ao usuário o nome do arquivo onde estão gravados os deslocamentos dos pontos nodais (a solução do sistema de equações) e o nome do arquivo onde serão gravados os esforços. Deve-se realizar tantas chamadas a esse método quantos forem os tipos de carregamentos atuantes na estrutura.

Para acrescentar uma barra ao modelo utiliza-se o método **Set_One_Bar**, que não requer nenhum parâmetro de entrada. Durante sua execução o método solicita ou usuário os dados necessários à inicialização de um objeto **TBarra** (ou **TBarra3d**, se for o caso), assim como: número do nó inicial, final, número que indica a posição do material na lista de materiais, etc. Após obter cada um desses dados, verifica na respectiva lista se o objeto associado à ele já existe armazenado e, caso não exista, inicializa-o para, posteriormente, associá-lo à barra. Assim, se na lista de nós não existe um nó que tenha o número igual ao número fornecido, o método inicializa um novo nó e, quando tiver todos os dados necessários à inicialização da barra, atribui-o à ela.

Para remover uma barra utiliza-se o método **Remove_One_Element** que solicitará ao usuário o número da barra à ser removida. O método fará uma pesquisa em todas as listas verificando se os objetos associados à barra em questão (nó inicial, cargas, material, etc) não estão também associados à outra(s) barra(s). Aquele que não estiver associado à outra barra, ou seja, for exclusivo da barra à ser removida, será também removido.

Os métodos **Chang_Bar_Ni** (altera o nó inicial de um objeto **TBarra** ou **TBarra3d**, substituindo o existente pelo que será fornecido), **Chang_Bar_Nf** (altera o nó final de um objeto **TBarra** ou **TBarra3d**), **Chang_Bar_Grup_Sec** (altera o grupo de seção transversal da barra), **Chang_Bar_Mat** (altera o tipo de material da barra) são utilizados sem parâmetro de entrada. Em tempo de execução, cada um solicita ao usuário que forneça o número do

objeto **TBarra** (ou **TBarra3d**) que sofrerá a mudança, bem como os dados do objeto da mudança. Por exemplo, para se alterar o nó inicial deve-se fornecer primeiramente o número da barra, depois o número do nó que substituirá o atual, após o que, o método verifica se existe um nó com esse número armazenado na lista de nós e, se não existir, exibe uma mensagem informando ao usuário que o nó não foi encontrado na lista e termina a execução do método (nesse caso deve-se criar o nó e depois executar novamete o método); caso exista, atribui-o ao elemento **TBarra** (ou **TBarra3d**) em substituição ao nó inicial.

O método **Main_Menu** exibe na tela as opções (transcritas à seguir) existentes neste trabalho para se analizar a estrutura:

Escolha :

[R] -> Ler Arquivo de dados.
[W] -> Criar arquivo de dados.

[B] -> Alterar dados de barras.
[L] -> Altera dados de cargas.
[M] -> Alterar dados de tipos de materiais.
[N] -> Alterar dados de nodes de estrutura.
[G] -> Alterar dados de secoes ou grupos de secoes transversais.
[S] -> Realiza a analise do modelo.
[C]-> Calcula os esforcos nas barras.
[T] -> Exibe na tela os resultados da analise.

[X] -> Encerra a analise do modelo.

As alternativas de alteração de dados ativa a os menus correspondentes que exibirão as opções mais específicas a serem escolhidas.

A possibilidade de se alterar a estrutura depois que ela foi armazenada pelo sistema, permitindo adicionar um elemento, alterar carregamentos, nós, etc, é muito importante na prática e no estudo teórico, pois permite verificar rapidamente quais os efeitos resultantes nas peças, obtidos devido a ocorrência de uma dessas modificações (diminuindo-se o vão de uma viga com a introdução de um pilar, por exemplo), bem como analisar com uma discretização mais apurada um determinado elemento ou região (região de descontinuidade, por exemplo) onde se obteve ou se supõe que haja uma concentração de tensões.

Todo o gerenciamento fica a cargo das funções da classe **TModel** e suas classes derivadas, portanto o programa principal implementado é relativamente simples, como mostra a listagem 6.13.

Listagem 6. 13 - Sequência de instruções do programa principal

```
void main (void)
{
   TModel * estrutura ;
   unsigned int espaco, aux ;
   aux = 0 ;
   while ( aux==0 )
         {
                cout<<"Digite qual o espaco da analise :\n"
                      <<"          2 = Plano\n"
                      <<"          3 = Tridimensional\n" ;
                cin>>espaco ;
                if ( espaco==2 )
                  {
                      estrutura = new TModel_With_Bar_2D( ) ;
                      estrutura->Main_Menu( ) ;
                      aux = 1 ;
                  }
                if ( espaco==3 )
                  {
                      estrutura = new TModel_With_Bar_3D( ) ;
                      estrutura->Main_Menu( ) ;
                      aux = 1 ;
                  }
                if ( aux==0 )
                  error("Opcao invalida. Tente novamente.\n",CONTINUAR) ;
         }
   delete estrutura ;
}
```

Como se percebe pelo código, as principais ações do programa principal são receber o espaço da análise, inicializar o objeto **TModel** de acordo com esse espaço, apresentar o menu de opções para se realizar a análise e, depois de obtido os resultados, remover o objeto **TModel**, retornando ao sistema o espaço de memória que foi utilizado. Parece bem simples mas a estrutura da interligação entre as diversas variáveis e dados que compõe o problema e sua solução é complexa.

Do modo como está desenvolvido o ambiente, para se tratar problemas que sejam discretizados com outros tipos de objetos derivados da classe **TElement** não será necessário redefinir as variáveis **protected** e alguns dos métodos que formam a classe **TModel**. Um elemento triangular, por exemplo, pode ser armazenado na mesma lista de elementos e responde aos mesmos métodos que um elemento de barra, porém de maneira diferente (polimorfismo). Será, portanto, necessário criar uma classe que gerencie a formação, destruição e a interligação desses novos objetos.

Para esse gerenciamento, essa nova classe será derivada de **TModel_Struct_Node_2D** ou **TModel_Struct_Node_3D**, por exemplo, herdando seus dados e métodos. Poderá também ser derivada de mais de uma dessas classes, caracterizando a herança multipla (uma outra vantagem da linguagem C++ orientada a objetos). Nesse caso, permitirá a análise do problema com diferentes tipos de elementos (quadrático com triangular ou triangular com barra, por exemplo). Tal fato se traduz em uma enorme vantagem para a ampliação do sistema num futuro projeto, pois as funções já testadas para gerenciamento do conjunto de objetos já criados não necessitarão ser reescritas, evitando-se a introdução de erros ao sistema e, também, podendo aumentar a abrangência da análise para alguns tipos de problemas (por exemplo, na maior discretização de uma região onde há concentração de tensões em apenas um dos elementos de barra que formam uma estrutura).

7 - EXEMPLOS

Neste capítulo são apresentados alguns dos exemplos que foram analisados com o programa desenvolvido. Essas mesmas estruturas foram também analisadas pelo programa comercial SAP-90 para que os resultados aqui obtidos tivessem um parâmetro de comparação e, a partir dessas comparações, se necessário, efetuar as correções.

Em todas as estruturas que serão apresentadas a seguir, convenciona-se aqui que o número dos nós da estrutura estarão apresentados dentro de círculos e o número das barras dentro de retângulos. As cargas e medidas de distâncias não estarão inscritas em nenhuma figura sendo que suas unidades as identificarão.

Por simplificação, para todas as barras dos exemplos seguintes, adota-se uma mesma seção transversal genérica, teórica, e que apresenta pequenos valores para as características geométricas, encontrando, assim, grandes valores para os deslocamentos. Tal seção tem as seguintes características:

Área = $30cm^2$.
Jz = $150cm^2$.
Jy = $60cm^2$.
Jt = $50cm^2$.

O material, também para todas as barras, apresenta módulo de elasticidade E = $2100tf/m^2$ e coeficiente de Poisson $\nu = 0,3$.

7.1 - Exemplo 1

Com a finalidade de facilitar a avaliação dos resultados obtidos em uma análise para que fosse possível realizar qualquer adequação, foram calculadas várias estruturas simples, aporticadas, como a apresentada pela figura 7.1. Essas estruturas são compostas por três barras formando um pórtico e possuem apenas um tipo de carga atuante, seja nodal ou distribuida.

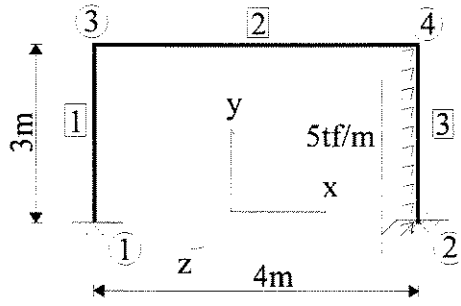

Figura 7.1 - Uma das estruturas aporticadas analisadas na fase de eliminação de erros.

Os resultados obtidos na análise desta estrutura pelo sistema aqui desenvolvido estão apresentados na tabela 7.1, e os obtidos com a utilização do programa SAP-90 estão apresentados na tabela 7.2.

Tabela 7.1 - Exemplo 1 - Resultados obtidos utilizando o programa desenvolvido.

Nó	u(x)	v(y)	w(z)	R(x)	R(y)	R(z)
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	-70.082	-0.413753	0.606533	0
4	0	0	-331.703	-1.37196	0.606533	0

Tabela 7.2 - Exemplo 1 - Resultados fornecidos pelo programa SAP-90

JOINT	U(X)	U(Y)	U(Z)	R(X)	R(Y)	R(Z)
1	.000000	.000000	.000000	.000000	.000000	.000000
2	.000000	.000000	.000000	.000000	.000000	.000000
3	.000000	.000000	-70.081986	-.413753	.606534	.000000
4	.000000	.000000	-331.703728	-1.371961	.606534	.000000

Essas tabelas demonstram a fácil comparação entre os resultados obtidos. Embora o exemplo seja de um pórtico plano simples, devido estar o carregamento em plano perpendicular ao do pórtico, foi realizada a análise como uma estrutura espacial.

7.2 - Exemplo 2

Após a fase de comparação e adequação dos resultados entre estruturas simples, procedeu-se a uma análise um pouco mais complexa. Para tanto foi proposto o estudo da estrutura apresentada pela figura 7.2. Os resultados obtidos com a utilização deste programa estão apresentado na tabela 7.3 e os obtidos com a utilização do programa SAP-90, na tabela 7.4.

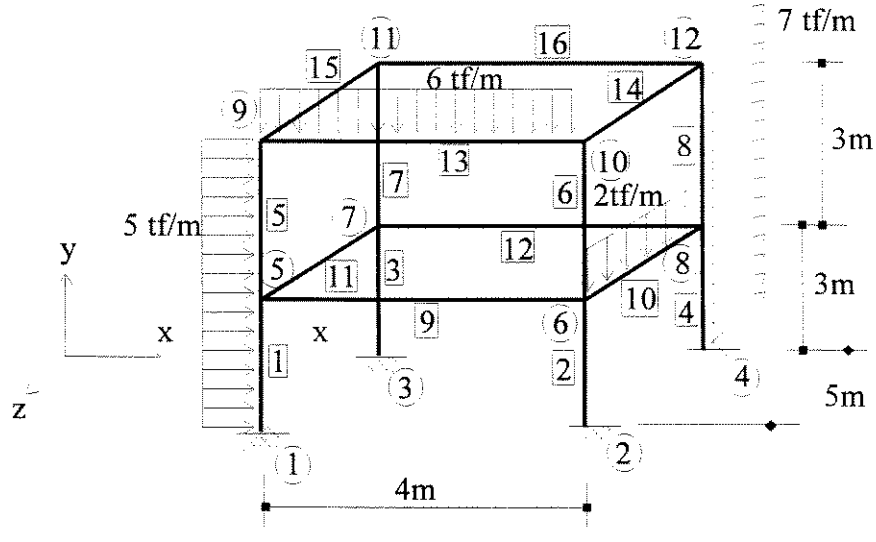

Figura 7.2 - Estrutura aporticada proposta para análise.

Tabela 7.3 - Exemplo 2 - Resultados obtidos utilizando o programa desenvolvido.

Nó	u(x)	v(y)	w(z)	R(x)	R(y)	R(z)
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	154.153	0.0215376	-67.7296	-0.144927	0.49472	-0.383814
6	154.121	-0.0952625	-318.248	-0.632267	0.494676	-0.430874
7	-21.3596	-0.0249518	-67.7297	-0.142035	0.494878	0.0583291
8	-21.3593	-0.063228	-318.336	-0.479239	0.494834	0.0571085
9	272.499	-0.0130421	-122.882	-0.0661259	0.777648	-0.272814
10	272.464	-0.160941	-490.906	-0.195649	0.777599	-0.0469836
11	-41.1721	-0.0330755	-122.882	-0.0621806	0.777718	0.0291583
12	-41.1723	-0.0691319	-490.939	-0.0654121	0.777668	0.0330186

Tabela 7.4 - Exemplo 2 - Resultados fornecidos pelo programa SAP-90.

JOINT	U(X)	U(Y)	U(Z)	R(X)	R(Y)	R(Z)
1	.000000	.000000	.000000	.000000	.000000	.000000
2	.000000	.000000	.000000	.000000	.000000	.000000
3	.000000	.000000	.000000	.000000	.000000	.000000
4	.000000	.000000	.000000	.000000	.000000	.000000
5	154.203645	.021580	-67.786180	-.145050	.494835	-.383973
6	154.171095	-.095272	-318.352947	-.632463	.494791	-.431033
7	-21.364775	-.024967	-67.786314	-.142158	.494992	.058342
8	-21.364518	-.063246	-318.440587	-.479435	.494949	.057122
9	272.609665	-.012983	-122.991811	-.066189	.777843	-.272916
10	272.573939	-.160956	-491.082158	-.195728	.777793	-.047086
11	-41.181775	-.033096	-122.991583	-.062244	.777912	.029165
12	-41.181952	-.069154	-491.114395	-.065491	.777863	.033025

A análise realizada neste caso representa um exemplo de estrutura realmente espacial. Observa-se que os valores obtidos com ambos os programas são praticamente os mesmos.

7.3 - Exemplo 3

O ambiente desenvolvido neste trabalho utiliza matrizes cheias para proceder a análise da estrutura. Assim, com o propósito de determinar qual a capacidade máxima de análise em um computador pessoal 486/DX-2 com 8Mb de memória RAM, partiu-se da estrutura apresentada na figura 7.2 e, com a utilização do método que acresenta barras, chegou-se a estrutura composta de 32 barras e 20 nós apresentada na figura 7.3. Os resultados obtidos com a utilização deste ambiente estão apresentados na tabela 7.5 e os fornecidos pelo programa SAP-90, na tabela 7.6.

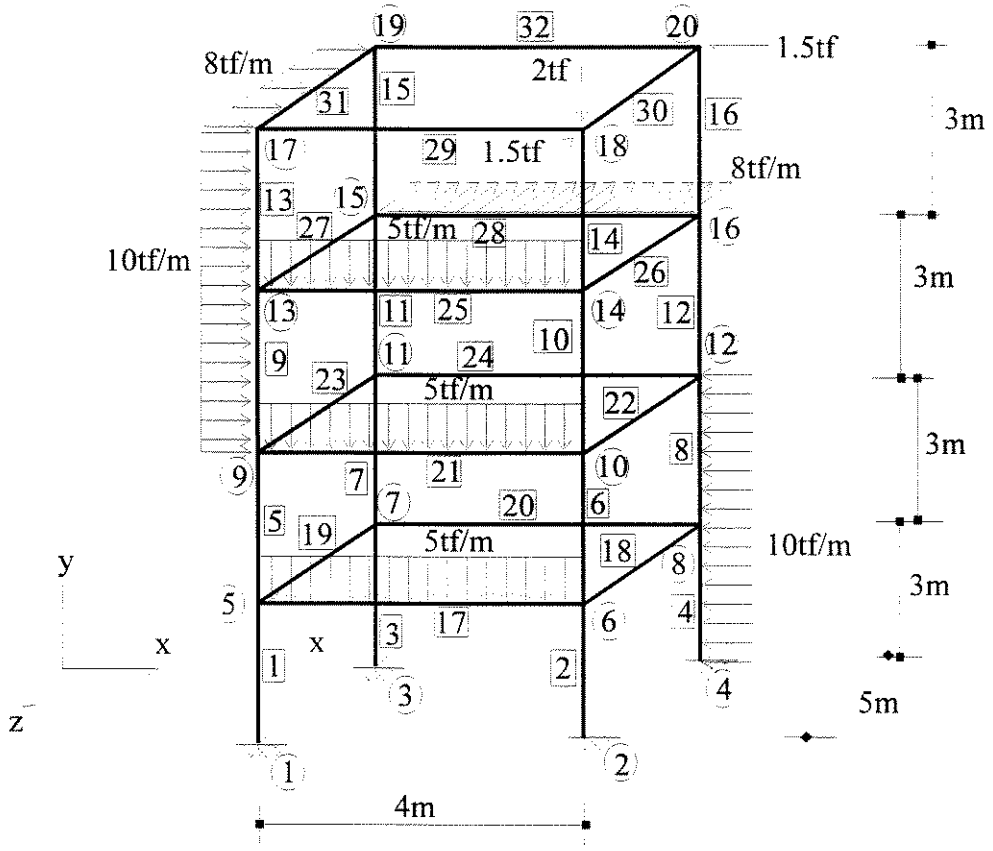

Figura 7.3 - Exemplo máximo cuja análise foi conseguida com a utilização de matriz cheia

Tabela 7.5 - Exemplo 3 - Resultados obtidos utilizando o programa desenvolvido.

Nó	u(x)	v(y)	w(z)	R(x)	R(y)	R(z)
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	364.792	0.301891	-59.4471	-0.189451	0.999082	-1.32022
6	364.802	-0.363177	-420.038	-1.01309	1.00586	-1.22027
7	-191.665	-0.0735946	-59.4471	-0.18943	1.00735	0.428824
8	-191.766	-0.160357	-420.038	-1.01307	0.997588	0.446773
9	933.553	0.495099	-177.517	-0.277145	2.1386	-1.40631
10	933.508	-0.608991	-970.502	-0.983064	2.19161	-1.24915
11	-280.545	-0.0843269	-177.517	-0.277262	2.21848	0.0373585
12	-280.592	-0.297019	-970.502	-0.983182	2.11173	-0.0439642
13	1387.36	0.566991	-316.463	-0.123339	2.81143	-0.853626
14	1387.24	-0.728956	-1423.45	-0.568633	3.13364	-0.716746
15	-247.947	-0.0754957	-316.526	-0.123351	3.57467	-0.100777
16	-247.934	-0.36254	-1423.51	-0.568637	2.3704	-0.0806347
17	1582.51	0.589746	-299.565	0.0404072	3.32544	-0.143704
18	1582.47	-0.754327	-1572.07	-0.137088	3.38502	-0.321063
19	-222.581	-0.0679999	-299.565	0.0405925	3.41358	-0.0357222
20	-222.585	-0.376942	-1572.07	-0.136889	3.29686	-0.0455583

Tabela 7.6 - Exemplo 3 - Resultados fornecidos pelo programa SAP-90.

JOINT	U(X)	U(Y)	U(Z)	R(X)	R(Y)	R(Z)
1	.000000	.000000	.000000	.000000	.000000	.000000
2	.000000	.000000	.000000	.000000	.000000	.000000
3	.000000	.000000	.000000	.000000	.000000	.000000
4	.000000	.000000	.000000	.000000	.000000	.000000
5	364.792153	.301891	-59.447028	-.189451	.999082	-1.320222
6	364.802568	-.363177	-420.038132	-1.013091	1.005859	-1.220275
7	-191.665377	-.073595	-59.447024	-.189430	1.007353	.428824
8	-191.765935	-.160357	-420.038128	-1.013069	.997588	.446774
9	933.553390	.495099	-177.517167	-.277145	2.138603	-1.406312
10	933.507707	-.608991	-970.502230	-.983064	2.191610	-1.249152
11	-280.545582	-.084327	-177.517180	-.277262	2.218485	.037359
12	-280.592536	-.297019	-970.502243	-.983183	2.111729	-.043964
13	.1387E+04	.5670	-.3165E+03	-.1233	.2811E+01	-.8536
14	.1387E+04	-.7290	-.1423E+04	-.5686	.3134E+01	-.7167
15	-247.947541	-.075496	-316.525990	-.123351	3.574671	-.100777
16	-.2479E+03	-.3625	-.1424E+04	-.5686	.2370E+01	-.8063E-01
17	.1583E+04	.5897	-.2996E+03	.4041E-01	.3325E+01	-.1437
18	.1582E+04	-.7543	-.1572E+04	-.1371	.3385E+01	-.3211
19	-222.580853	-.068000	-299.565258	.040593	3.413579	-.035722
20	-.2226E+03	-.3769	-.1572E+04	-.1369	.3297E+01	-.4556E-01

7.4 Exemplo 4

Apresenta-se a seguir um exemplo de combinações entre tipos de carregamentos. Para tanto foi utilizada a mesma estrutura do item anterior sujeita agora a dois tipos de carregamentos (mostrados no esquema da figura 7.4).

Para a análise foram considerados dois tipos de combinações entre os carregamentos. Na primeira o carregamento 1 tem fator de multiplicação igual a 1,4 e o carregamento 2 tem peso 0.8 e os resultados obtidos estão apresentados na tabela 7.7. Na segunda combinação ambos os tipos de carregamentos possuem fatores de multiplicação iguais a 1,2 e os resultados obtidos estão na tabela 7.8.

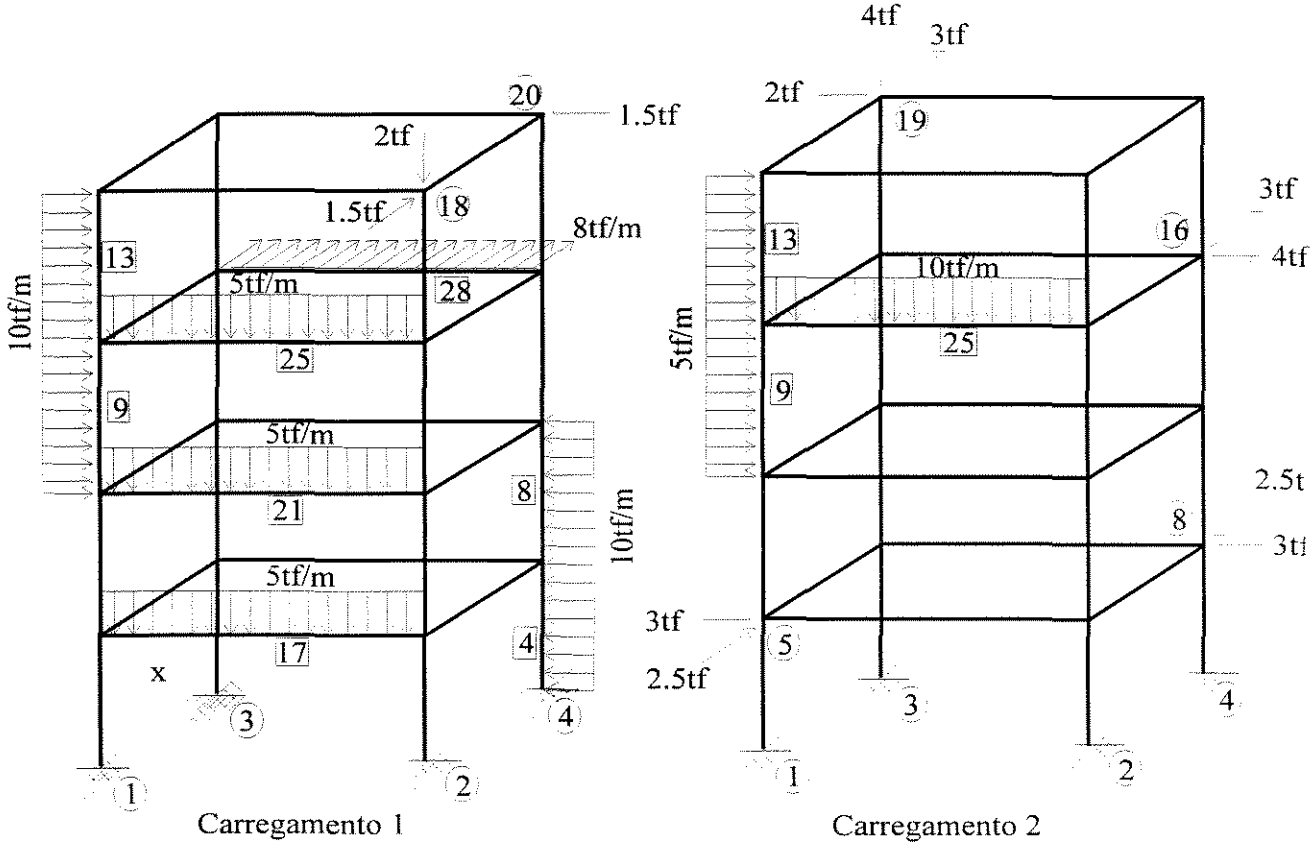

Figura 7.4 - Exemplo de análise considerando-se combinação de carregamentos

Tabela 7.7 - Exemplo 4 - Resultados obtidos para a primeira combinação entre os carregamentos.

Nó	u(x)	v(y)	w(x)	R(x)	R(y)	R(z)
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	367.123	0.228999	-179.017	-0.480788	1.08586	-1.34598
6	367.124	-0.431714	-552	-1.3535	1.09535	-1.18884
7	-270.177	-0.15631	-179.009	-0.480755	1.09744	0.595247
8	-270.327	-0.221929	-551.992	-1.35347	1.08377	0.620824
9	931.623	0.347707	-455.262	-0.584838	2.28232	-1.38771
10	931.592	-0.751845	-1296.55	-1.34349	2.35658	-1.25053
11	-384.138	-0.209638	-455.262	-0.585016	2.39415	0.0194034
12	-384.203	-0.4148	-1296.55	-1.34369	2.24474	-0.0958716
13	1383.65	0.342955	-746.495	-0.327928	2.89564	-0.997243
14	1383.52	-0.954004	-1916.3	-0.766154	3.34677	-0.5693
15	-322.308	-0.225175	-746.585	-0.327955	3.96419	-0.169433
16	-322.28	-0.506641	-1916.39	-0.766157	2.27825	-0.138545
17	1578.18	0.36978	-787.909	-0.0172449	3.42504	-0.0907936
18	1578.13	-0.98078	-2105.25	-0.16917	3.50847	-0.372458
19	-280.505	-0.234426	-787.918	-0.0170077	3.54842	-0.056543
20	-280.517	-0.526008	-2105.24	-0.168866	3.38503	-0.0717271

Tabela 7.8 - Exemplo 4 - Resultados obtidos para a segunda combinação entre os carregamentos.

Nó	u(x)	v(y)	w(z)	R(x)	R(y)	R(z)
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	222.272	0.0718092	-215.371	-0.551509	0.72982	-0.830389
6	222.265	-0.320402	-450.436	-1.11959	0.737947	-0.684631
7	-232.899	-0.168498	-215.359	-0.551479	0.739741	0.507346
8	-233.032	-0.188627	-450.424	-1.11956	0.728021	0.529558
9	556.956	0.0759792	-523.904	-0.628544	1.4991	-0.815461
10	556.951	-0.57918	-1072.42	-1.13138	1.56278	-0.751129
11	-324.008	-0.238991	-523.904	-0.628705	1.59496	-0.00419844
12	-324.065	-0.354957	-1072.42	-1.13157	1.46693	-0.103921
13	826.414	0.00413996	-836.009	-0.381319	1.81354	-0.727295
14	826.332	-0.774378	-1594.73	-0.637783	2.20025	-0.208573
15	-260.701	-0.270318	-836.086	-0.381349	2.72945	-0.163267
16	-260.67	-0.43374	-1594.82	-0.637785	1.28438	-0.135063
17	942.486	0.023892	-913.487	-0.0623957	2.14483	-0.0067056
18	942.446	-0.791692	-1744.41	-0.130357	2.21635	-0.269579
19	-220.882	-0.290963	-913.501	-0.0622068	2.25058	-0.0525719
20	-220.896	-0.449817	-1744.4	-0.130081	2.11054	-0.0664957

7.4 Exemplo 5

Uma das vantagens proporcionadas pela filosofia de programação orientada por objetos é permitir a utilização de códigos gerados por outros pesquisadores praticamente sem que sejam feitas restrições a estrutura de dados e, também, com pouco tempo gasto no entendimento da forma de utilização desse código. Objetivando demonstrar aqui essa vantagem, utiliza-se para solução de sistemas de equações lineares um conjunto de classes desenvolvido por MANDUJANO[1995]. Essas classes foram implementadas com a finalidade de permitir a utilização de vários tipos de matrizes, das quais, devido às características do trabalho aqui desenvolvido, aquelas que permitem criar objetos do tipo matriz ***skyline*** serão adaptadas a este trabalho.

Ainda, demonstrando a possibilidade da ampliação deste trabalho por outros pesquisadores que em princípio não terão acesso ao código fonte, a implementação do armazenamento da matriz de rigidez da estrutura em forma de matriz ***skyline***, a solução do sistema de equações e a gravação dos resultados (deslocamentos) em arquivo, foram realizadas simulando-se que apenas os arquivos que contém as definições das classes (arquivos de cabeçalho) são conhecidos. Assim sendo, nenhuma parte do código já desenvolvido (seja deste trabalho ou do que trata de matrizes) foi aberta e reescrita.

A fim de atender às especificações acima, foi desenvolvida uma nova classe, denominada **TModel_Integrated**, cujo arquivo de cabeçalho está em parte transcrito na listagem 7.1. Essa classe é devivada da classe **TModel_With_Bar_3D,** herdando desta seus métodos, e contém a inclusão dos arquivos que permitem criar o objeto matriz ***skyline*** (arquivos **tsimmat.h**, **tfullmat.h** e **tskylmat.h**). Desta forma, apenas os métodos (herdados) que de alguma utilizam ou se relacionam com a matriz de rigidez da estrutura foram reescritos. Assim, temos os seguintes métodos redefinidos:

- **Set_Cond_Contour** → definido na área **protected** da classe e é o responsável por incluir na matriz de rigidez as condições de contorno (graus de liberdade restritos e deslocamentos impostos) da estrutura.

- **Fill_Mat_Glob** → método que preenche a matriz de rigidez global processando as contribuições das barras. Também definido na área **protected**.

- **Fill_Vet_Glob** → método que preenche o vetor de cargas global somando, na coordenada correspondente, a contribuição das cargas nodais e das cargas atuantes nas barras (cargas nodais equivalentes). Na chamada a esse método deve-se fornecer como parâmetro de entrada o número do tipo de carregamento para o qual se esta calculando os deslocamentos (resolvendo o sistema). Esse método foi reescrito porque na chamada ao método que soluciona o sistema de equações associado ao objeto do tipo **TSkylMatrix** (classe do trabalho acima referido desenvolvida para representar matrizes de banda ***skyline***), deve-se passar como parâmetro um ponteiro para um objeto da classe **TFMatrix** (classe do mesmo trabalho que representa uma matriz cheia). Esse último objeto está associado, neste trabalho, ao vetor de carga globais. Portanto, o método de mesmo nome da classe **TModel_With_Bar_3D** não é compatível nesta nova classe por armazenar o vetor de cargas em um objeto da classe **TVetFloat**.

- **Save_Global_Result** → método que salva os resultados da solução do sistema de equações em um arquivo.

- **Solve** → Faz a chamada ao método que preenche a matriz de rigidez da estrutura e ao método que implementa nela as condições de contorno. A seguir, armazena a matriz assim obtida em um arquivo, e, para cada um dos tipos de carregamentos que a estrutura apresenta, chama o método que preenche o vetor de carga (passando o número do tipo de carregamento), preenche a variável que representa a matriz de rigidez com os dados lidos

do arquivo onde a matriz foi armazenada, faz a chamada ao método que soluciona o sistema de equações (método **Solve_Cholesky** da classe **TSkylMatrix**) e salva os resultados solicitando ao usuário o nome do arquivo onde eles serão armazenados.

Listagem 7.1 - Parte do arquivo de cabeçalho da classe **TModel_Integrated**

```
#include "tsimmat.h"
#include "tskylmat..h"
#include "tfullmat.h"
#include "modbar3d.h"
#include "vetint.h"
                    class TModel_Integrated : public TModel_With_Bar_3D
{
 protected :
        TSkylMatrix * fmatrix_global ;
        TFMatrix * fvetor_global ;
        virtual void Set_Cond_Contour( ) ;
        virtual void Fill_Mat_Glob( ) ;
        virtual void Fill_Vet_Glob(unsigned int number_of_type_load) ;
        virtual void Save_Global_Result( ) ;
 public :
        TModel_Integrated( ) ;
        virtual ~TModel_Integrated( ) ;
        virtual void Solve( ) ;
} ;
```

Estão definidas na área **protected** desta classe duas variáveis: um ponteiro para um objeto da classe **TSkylMatrix** denominado **fmatriz_global** que representa a matriz de rigidez da estrutura (armazenada na forma de matriz ***skyline***) e um ponteiro para um objeto da classe **TFMatrix**, denominado **fvetor_global**, ao qual serão atribuidos os valores que compõem o vetor de cargas globais (armazenado na forma de uma matriz coluna). Essas variáveis são inicializadas no método construtor da classe.

Com a introdução dessa nova classe foi possível analisar, por exemplo, a estrutura apresentada na figura 7.5. Os resultados obtidos estão apresentados na tabela 7.9, e os conseguidos com a utilização do programa SAP-90 para comparação, na tabela 7.10.

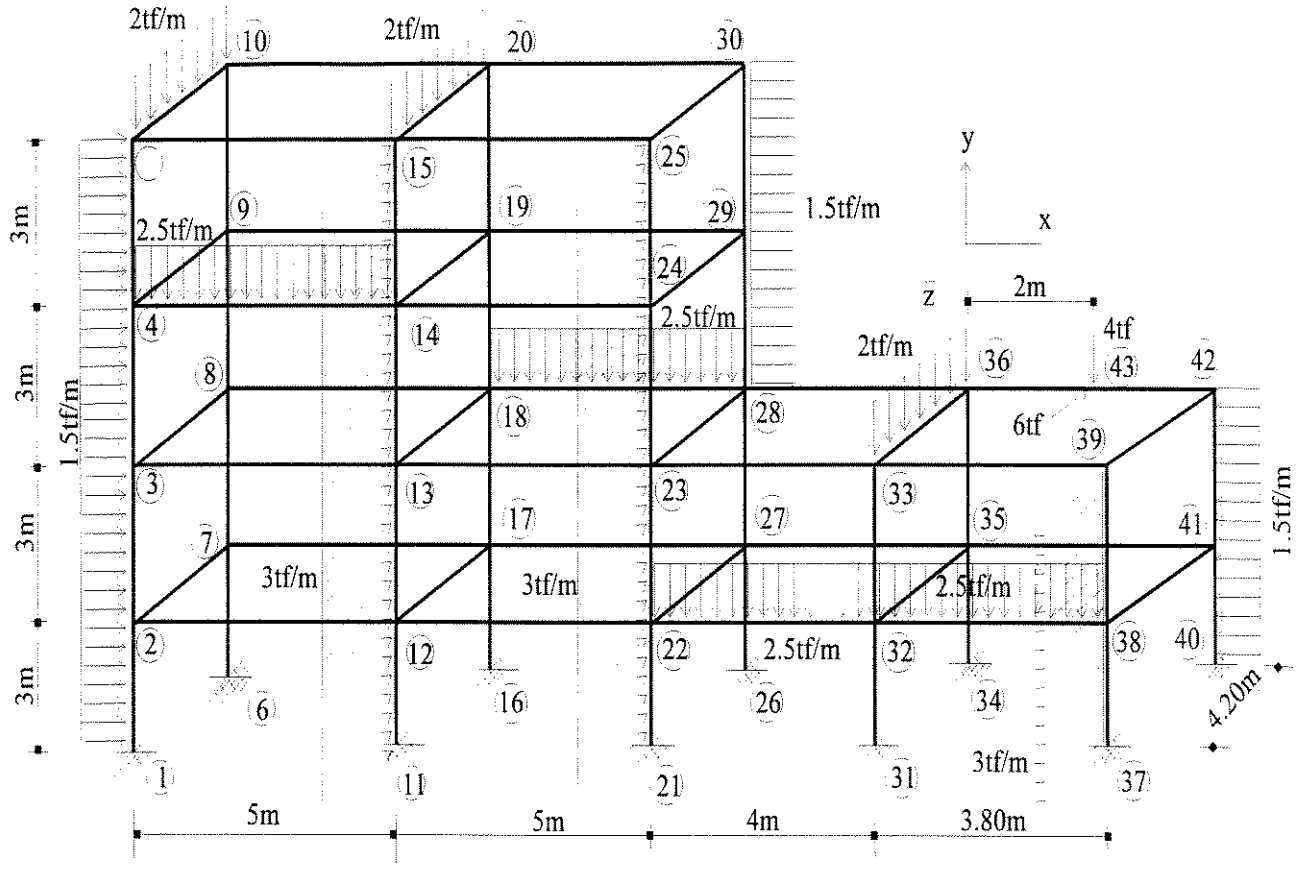

Figura 7.5 - Exemplo de estrutura analisada utilizando matriz *skyline*

Na figura 7.5 não estão marcadas os números da barras. Assim se procedeu para facilitar a visualização da estrutura, pois caso esses número fossem adicionados ao desenho, tornariam-no pesado e confuso.

Tabela 7.9 - Exemplo 5 - Resultados obtidos utilizando o programa desenvolvido.

Nó	u(x)	v(y)	w(z)	R(x)	R(y)	R(z)
1	0	0	0	0	0	0
2	27.1859	0.0482147	-131.69	-0.278564	0.137913	-0.0964363
3	67.8586	0.062065	-280.764	-0.22656	0.254765	-0.135353
4	137.222	0.0436826	-379.818	-0.110228	0.367082	-0.219489
5	178.975	0.0318057	-425.992	-0.14144	0.41608	-0.0323269
6	0	0	0	0	0	0
7	19.7325	-0.0773816	-131.69	-0.279589	0.137554	-0.0769069
8	48.2815	-0.130803	-280.763	-0.220569	0.253432	-0.0553577
9	64.0318	-0.166803	-379.823	-0.14416	0.366755	-0.0265541
10	67.1121	-0.191094	-425.988	0.05035	0.416009	-0.00037962
11	0	0	0	0	0	0
12	27.1496	0.0579668	-268.426	-0.521587	0.109659	-0.0679088
13	67.7974	0.0627987	-526.69	-0.361807	0.222875	-0.127609
14	137.186	0.0319374	-674.231	-0.157932	0.325455	-0.0674648
15	178.951	0.0154519	-720.989	-0.0823817	0.388718	-0.0657878
16	0	0	0	0	0	0
17	19.7401	-0.161534	-268.396	-0.522182	0.110443	-0.0480461
18	48.3014	-0.269332	-526.66	-0.358153	0.227534	-0.093667
19	64.0498	-0.309029	-674.204	-0.17839	0.326088	-0.00786133
20	67.1212	-0.332107	-720.971	0.0340861	0.38879	-0.00858413
21	0	0	0	0	0	0
22	27.1133	0.104006	-257.049	-0.516305	-0.152914	-0.094475
23	67.692	0.175782	-533.182	-0.460964	-0.270629	-0.108274
24	137.161	0.20114	-774.334	-0.311303	0.145459	-0.171812
25	178.943	0.20405	-873.717	-0.0412577	0.259382	-0.06303
26	0	0	0	0	0	0
27	19.7598	-0.167015	-257.019	-0.517342	-0.155898	-0.0617189
28	48.3411	-0.283472	-533.151	-0.458506	-0.295983	-0.00796961
29	64.0781	-0.323413	-774.302	-0.300492	0.143296	-0.0387793
30	67.1393	-0.330592	-873.704	-0.10975	0.259187	0.0196038
31	0	0	0	0	0	0
32	27.096	-0.0322779	-99.5029	-0.199531	-0.275407	-0.070132
33	67.6103	-0.0395773	-212.949	-0.221234	-0.520145	-0.0372556
34	0	0	0	0	0	0
35	19.7594	-0.0604445	-99.5082	-0.230071	-0.266741	-0.0455559
36	48.3489	-0.100358	-212.957	-0.0384084	-0.397454	-0.0569559
37	0	0	0	0	0	0
38	27.088	-0.0324252	31.4276	0.0082828	-0.142871	-0.0704057
39	67.5793	-0.0333404	-2.32104	-0.104652	-0.178912	-0.08455
40	0	0	0	0	0	0
41	19.7439	-0.0300617	31.3955	-0.00569072	-0.157684	-0.0737265
42	48.3415	-0.0528105	-2.34395	-0.0263402	-0.363373	-0.0357764
43	48.345	-4.44579	-113.123	-0.0320567	-0.650787	0.0267341

Tabela 7.10 - Exemplo 5 - Resultados fornecidos pelo programa SAP-90.

JOINT	U(X)	U(Y)	U(Z)	R(X)	R(Y)	R(Z)
1	.000000	.000000	.000000	.000000	.000000	.000000
2	27.185884	.048215	-131.690206	-.278564	.137913	-.096436
3	67.858572	.062065	-280.764432	-.226560	.254765	-.135353
4	137.221777	.043683	-379.818249	-.110228	.367083	-.219489
5	178.974734	.031806	-425.992523	-.141440	.416080	-.032327
6	.000000	.000000	.000000	.000000	.000000	.000000
7	19.732477	-.077382	-131.690368	-.279589	.137554	-.076907
8	48.281471	-.130803	-280.763478	-.220569	.253432	-.055358
9	64.031810	-.166803	-379.823444	-.144160	.366755	-.026554
10	67.112072	-.191094	-425.988104	.050350	.416009	-.000380
11	.000000	.000000	.000000	.000000	.000000	.000000
12	27.149580	.057967	-268.425708	-.521587	.109659	-.067909
13	67.797373	.062799	-526.690021	-.361807	.222875	-.127609
14	137.186464	.031937	-674.231343	-.157932	.325455	-.067465
15	178.951116	.015452	-720.988833	-.082382	.388718	-.065788
16	.000000	.000000	.000000	.000000	.000000	.000000
17	19.740070	-.161534	-268.395844	-.522182	.110443	-.048046
18	48.301414	-.269332	-526.659709	-.358153	.227534	-.093667
19	64.049831	-.309029	-674.204522	-.178390	.326088	-.007861
20	67.121190	-.332107	-720.971145	.034086	.388791	-.008584
21	.000000	.000000	.000000	.000000	.000000	.000000
22	27.113261	.104006	-257.049122	-.516305	-.152914	-.094475
23	67.692003	.175782	-533.182347	-.460964	-.270629	-.108274
24	137.161226	.201140	-774.333883	-.311303	.145459	-.171812
25	178.943361	.204050	-873.717539	-.041258	.259383	-.063030
26	.000000	.000000	.000000	.000000	.000000	.000000
27	19.759759	-.167015	-257.019088	-.517342	-.155898	-.061719
28	48.341059	-.283472	-533.150940	-.458506	-.295983	-.007970
29	64.078072	-.323413	-774.301884	-.300492	.143296	-.038779
30	67.139312	-.330592	-873.704149	-.109750	.259187	.019604
31	.000000	.000000	.000000	.000000	.000000	.000000
32	27.096029	-.032278	-99.502906	-.199531	-.275407	-.070132
33	67.610264	-.039577	-212.949030	-.221234	-.520145	-.037256
34	.000000	.000000	.000000	.000000	.000000	.000000
35	19.759375	-.060444	-99.508221	-.230071	-.266741	-.045556
36	48.348922	-.100358	-212.956586	-.038408	-.397454	-.056956
37	.000000	.000000	.000000	.000000	.000000	.000000
38	27.087964	-.032425	31.427562	.008283	-.142871	-.070406
39	67.579313	-.033340	-2.321055	-.104652	-.178912	-.084550
40	.000000	.000000	.000000	.000000	.000000	.000000
41	19.743850	-.030062	31.395486	-.005691	-.157684	-.073727
42	48.341458	-.052811	-2.343965	-.026340	-.363373	-.035776
43	48.344994	-4.445788	-113.123091	-.032057	-.650787	.026734

Confirma-se com esse exemplo que a possibilidade de combinar dois ou mais códigos existentes, ampliando a gama de problemas possíveis de serem analisados, apresenta vantagens com a utilização da orientação por objetos, dentre as quais pode-se salientar as seguintes:

- menor tempo gasto na implementação da nova potencialidade, pois não houve necessidade de se proceder ao desenvolvimento, excução de testes e eliminação de erros do código que representa as matrizes;
- possibilidade reduzida de introduzir erros em código já testado (impossibilidade neste caso, pois o código não foi aberto), no qual praticamente não há restrições à estrutura de dados;
- qualquer método que precise ser desenvolvido ou alterado, de modo a adaptar o código que se deseja introduzir, pode ser implementado em uma sub-classe. Isso permite a inclusão de qualquer conjunto de classes praticamente sem restrições;

Note-se que o exemplo apresentado pela figura 7.5 não representa a potencialidade máxima do ambiente utilizando matriz de banda ***skyline***, pois a intenção com esse exemplo não foi a de verificar esse máximo, mas sim a de comprovar as vantagens anteriormente referidas que são oferecidas pela programação orientada por objetos.

8 - CONCLUSÕES

8.1 - INTRODUÇÃO

Foi desenvolvido um sistema computacional baseado na filosifia da linguagem orientada por objetos. Esse sistema é composto por uma biblioteca de classes de objetos relacionados a estruturas aporticadas tridimensionais que permitem a análise linear desse tipo de estrutura com o emprego do método dos elementos finitos. Devido à modularidade que essa biblioteca apresenta, serve de base para um sistema mais complexo, pois a ela podem ser adicionadas novas teoriais e idéias sem que seja necessário reescrever o código já existente. Isso é conseguido, conforme o procedimento comentado no Capítulo 7, através da criação de classes derivadas das existentes, especializadas com os acréscimos de dados e métodos. Assim, o sistema permite a quem vier a ampliá-lo, dedicar-se exclusivamente à implementação da nova potencialidade, conseguindo uma otimização de tempo e esforço.

Exemplos foram realizados demonstrando o funcionamento do sistema e a facilidade de sua ampliação por outros pesquisadores. Comparações dos resultados obtidos foram feitas de modo a trazer maior credibilidade ao código que foi aqui implementado, permitindo que as futuras ampliações a serem realizadas tenham um patamar de partida confiável.

8.2 - AVALIAÇÃO DA UTILIZAÇÃO DA OOP

O desenvolvimento do programa apresentado nos capítulos 5 e 6 demonstrou as vantagens da filosofia de programação orientada por objetos. Alterações nas especificações de projeto (muito comuns no desenvolvimento de *softwares*) foram feitas de modo mais fácil do seriam caso fosse utilizada a linguagem procedural. Isso ocorreu basicamente devido a essas alterações estarem localizadas em algumas das classes do sistema, não havendo, portanto, a necessidade de se proceder às alterações a partir do início do programa e não tendo que modificar outras partes do código já estável. Essa é uma característica da propriedade do encapsulamento, a qual permitiu escrever um código que tem uma manutenção mais fácil, com menor probabilidade de introdução de erros.

A propriedade da herança foi útil na implementação do código por permitir uma economia de tempo e esforço considerável ao se fazer as sub-classes herdarem métodos e dados de super-classes. Tais métodos precisaram ser implementados uma única vez, sendo que para as sub-classe eles foram complementados de acordo com as especificações que cada uma delas necessitou. Isso também amplia a quantidade de classes semelhantes que podem ser abrangidas e permite o crescimento indefinido dos tipos de problemas que possam vir a ser analisados pelo sistema (problemas de transferência de calor, escoamento de fluidos, eletromagnetismo, etc), não se restringindo apenas ao campo da análise estrutural.

O conceito de classe e a propriedade da herança também permitiram uma melhora no gerenciamento dos dados e a modularização do sistema. Assim, como apenas o objeto pode chamar um método que altere os dados a ele associados, diminuem as possibilidades de o programador cometer um engano fazendo com que um dado receba um valor que não era intencionalmente a ele endereçado ou que armazene uma inconsistência. Já a modularização permitiu dividir

o códogo em blocos, geralmente individuais, facilitando a implementação e ampliação do ambiente.

Ainda com relação à modularização, na fase de testes notou-se outra vantagem da linguagem OOP sobre a procedural. Como as classes representam conjunto de objetos geralmente independentes, seus códigos podem ser testados e avaliados antes de se haver completado todo o projeto. Isso facilita sobremaneira a localização e eliminação de erros, pois não há a necessidade de se efetuar a procura por todo o código, restringindo-a a umas poucas linhas.

Observou-se também que a modularização favorece o desenvolvimento de *software* por um grupo de pesquisadores, pois cada participante do grupo pode implementar um determinado tipo de procedimento concomitante e independentemente dos demais. Isso leva a uma maior rapidez na execução do produto final.

Essas propriedades, aliadas ao polimorfismo, permitiram desenvolver mais as classes que já haviam sido criadas para o espaço bidimensional, ampliando-as com a derivação de classes mais específicas para atender aos objetos mais especializados (barras e nós no espaço tridimensional, por exemplo) , introduzindo uma nova potencialidade (análise tridimensional) ao sistema existente, não necessitando, entretando, que fosse aberto o código já testado e estável. Dessa forma, o objetivo primeiro deste projeto, que é um ambiente computacional orientado por objetos para análise de estruturas tri-dimensionais, foi alcançado sem que fosse perdida a possibilidade de se efetuar uma análise no plano.

8.3 - ANÁLISE DE RESULTADOS DE SIMULAÇÕES

A introdução da utilização de um código gerado por um outro pesquisador independente (matriz de banda simétrica) mostrou requerer pouco tempo no entendimento de como utilizar o objeto a ser introduzido, e, também, não houve restrição à estrutura de dados. A propriedade da herança permitiu a implementação da nova potencialidade ao sistema completamente sem a possibilidade de introdução de erros, pois nenhum dos códigos já testados teve de ser aberto para uma readaptação, bastando criar uma nova classe, mais específica, que representasse a nova potencialidade., mostrando assim, a vantagem que a orientação por objetos oferece na reutilização ou na ampliação de um código já existente.

É importante salientar que o método dos elementos finitos permite simular o comportamento de um sistema físico complexo através de uma transformação. Essa transformação aproxima as equações parciais que regem o problema em um conjunto de equações algébricas às quais se pode aplicar as propriedades das matrizes e solucionar o problema por computador.

Os problemas numéricos analisados por este ambiente computacional, foram também calculados utilizando-se o programa comercial **SAP-90**. Os resultados numéricos fornecidos pelas duas análises apresentaram diferenças minimas, indicando uma boa precisão e dando confiabilidade ao sistema aqui desenvolvido. Ainda, para os exemplos analisados, o tempo que cada um dos programas levou para efetuar o processamento foi praticamente o mesmo. Entretanto este dado é subjetivo, pois não foram efetuadas medições e os exemplos não correspondem a estruturas muito complexas.

8.4 - SUGESTÕES PARA TRABALHOS FUTUROS

Apresenta-se a seguir alguns recursos especiais que podem ser implementados a esse sistema através da criação de classes mais específicas, derivadas das aqui existentes. São eles:

- Permitir integração por pontos de Gauss

Pode ser criada uma classe que permita a substituição da integração de uma função por uma operação matemática entre os valores que essa função assume em determinados pontos notáveis (denominados pontos de Gauss). Utiliza-se essa nova classe, por exemplo, na obtenção do carregamento nodal equivalente de uma carga que não tenha sido aqui implementada e cuja lei de variação siga uma determinada expressão matemática. Pode-se, então, através da criação de uma sub-classe da classe **TLoad** que contenha um objeto dessa classe representativa dos pontos de Gauss, obter o carregamento nodal equivalente desse novo tipo de carga.

- Análise de elemento de barra com seção variável

Utilizando-se um procedimento semelhante ao descrito no item anterior pode-se permitir que o usuário faça análise de elemento com seção transversal variando segundo uma função qualquer (viga em mísula, por exemplo). A implementação desse procedimento poderá ser feita em uma classe que tem a classe **TGenericSecao** como raiz e terá seus métodos desenvolvidos de modo a considerar as alterações nas características geométricas da seção transversal, traduzindo-as na matriz de rigidez do elemento, evitando-se, assim, uma revisão em todo o código já testado.

- Implementar um procedimento que permita analisar o efeito de conexões semi-rígidas

Considerando-se o método aplicado por BLANDFORD[1994], pode-se levar em consideração os efeitos provocados por conexões semi-rígidas implementando-se uma classe derivada da classe **TBarra** (ou **TBarra3d**, conforme o caso) que tenha a característica de o comprimento do objeto ser nulo, alterando-se, desta forma, a matriz de rigidez do elemento. Provávelmente ter-se-á que escrever apenas os métodos que calculam a matrix de rigidez do elemento segundo as coordenadas locais, a matriz de rotação, assim como a função construtora e destrutora.

- Implementar métodos para armazenar e ler o estado atual dos diversos objetos

Pode-se implementar métodos (nestas classes ou em sub-classes) para que os diversos objetos de diversas classes tenham a atual situação de seus dados armazenada ou lida de um arquivo. Esses métodos poderão ser criados em cada uma das classes que terão objetos a serem armazenados, não necessitando alterar nenhuma das funções já existentes e testadas ou apenas serem implementados na classe que gerencia toda a análise. Tal artifício será muito útil na entrada manual de dados; no processamento de um problema longo; e também quando, por algum motivo, o computador for desligado ou o usuário tiver que se ausentar.

Pode-se também utilizar esses métodos na implementação de uma função **Undo** (utilizada por PESQUERA et all[1983] e PAULINO[1988] no desenvolvimento de pré-processadores) que permita ao usuário o retorno ao estágio anterior ao que foi executado caso este resulte em um erro fatal ou caso os resultados obtidos não tenham sido satisfatórios, economizando um tempo considerável ao usuário, pois assim este não terá que processar novamente etapas concluidas e de resultados aceitáveis.

- Utilização da matriz de rigidez em blocos

Para poder análisar estruturas mais complexas faz-se necessária a implementação de um procedimento que permita trabalhar com matrizes de rigidez maiores que as conseguidas neste trabalho. Para tanto sugere-se a criação de uma classe derivada da classe **TSqMatrix** aqui desenvolvida que grave a matriz de rigidez da estrutura em um arquivo e, a partir deste arquivo, obtenha blocos da matriz para proceder aos cálculos (decomposição da matriz e solução dos sistemas de equações). Deve-se também criar uma classe derivada da classe que gerencia o sistema (**TModel_With_Bar_2d** ou **TModel_With_Bar_3d**), que contenha um objeto dessa nova classe de matrizes e que provavelmente terá apenas os métodos **Set_Cond_Contour**, **Fill_Mat_Glob** e **Solve** reescritos para proporcionar acesso à essa nova potencialidade. Esse procedimento ampliaria em muito o tamanho permitido para a estrutura.

- Análise não-linear

Através de sub-classes da classe **TMaterial** que permitam avaliar os valores do módulo de elasticidade e do módulo tangente através de processos incrementais pode-se obter a matriz de rigidez de objetos **TBarra3d**, **TBarra2d** ou derivados se necessário, nas quais os efeitos da não-linearidade física ou geométrica estejam presentes. A classe que gerencia o modelo também deverá ter sua sub-classe que permita esse tipo de análise.

- Análise dinâmica

Pode-se criar sub-classes das classes **TBarra2d** ou **TBarra3d** que considerem na montagem da matriz de rigidez os efeitos das solicitações harmônicas, assim como sub-classes da classe **TLoad** devem ser implementadas para representar esse tipo de carregamento. Também será necessária a criação de uma classe derivada da que gerencia o modelo e que acrescente nele o estudo das vibrações.

Enfim, muitos outros trabalhos tais como a criação de uma biblioteca de classes cujos objetos representem elementos utilizados na análise de placas e cascas, a criação de pré e pós-processadores que permitam uma interface mais amigável com o usuário, classes que gerem o modelo (malha), classes que permitam estudos de otimização, sub-estruturação, dimensionamento e até detelhamento das peças estruturais, podem servir-se deste sistema e ampliá-lo, pois o programa aqui desenvolvido é apenas o início.

APENDICE A

ROTEIRO PARA ELABORAÇÃO DE ARQUIVO DE DADOS

Um arquivo de dados para o ambiente aqui desenvolvido deve ser redigido em um editor de textos que grave o arquivo em formato ASCI. Por exemplo: edit do DOS, editor do turbo Pascal ou do Borland C++.

Importante:

- Não utilize vírgula para separar os dados.
- As casas decimais devem ser separadas da parte inteira por ponto e não por vírgula.
- Os dados relativos a análise tridimensional não deverão ser digitados (nem mesmo o valor zero) quando a análise se der no plano.
- Quando não existir um determinado valor para o dado, digite zero.

Os dados deverão ser digitados conforme estão descriminados a seguir:

Número de nós do modelo

Para cada nó digite:

- número do nó;
- coordenada X;
- coordenada Y;
- coordenada Z.

Número de nós com restrição a deslocamentos

Para cada nó com restrição digite:

- número do nó,
- restrição à translação em X,
- restrição à translação em Y,
- restrição à translação em Z,
- restrição à rotação em X,
- restrição à rotação em Y,
- restrição à rotação em Z.

Número de nós com recalques de apoio

Para cada nó com recalque forneça:

- número do nó,
- recalque de translação em X,
- recalque de translação em Y,
- recalque de translação em Z,
- recalque de rotação em X,
- recalque de rotação em Y,
- recalque de rotação em Z.

Número de seções transversais

Para cada seção digite:

- Tipo de seção transversal → as opções são : C para cantoneira de abas iguais; L para cantoneira de abas desiguais; H para perfil H; I para perfil I ; U para perfil U e Q para outra seção qualquer.

- Os dados da seção → para cada tipo deve-se

tipo C → fornecer o valor da aba e da espessura da cantoneira.

tipo L → fornecer o valor da altura, o valor da aba e o valor da espessura da cantoneira.

tipos H, I e U → fornecer o valor da altura e o peso por metro linear do perfil.

tipo Q → fornecer o valor da área, do momento de inércia com relação ao eixo Z, o momento de inércia com relação ao eixo Y, a constante de torção e o peso por metro linear da seção.

Número de grupos de seções transversais

Para cada grupo forneça o número da seção a ele correspondente (apenas digite o número da seção. Não digite o número do grupo)

Número de materiais que compõe o modelo

Para cada material forneça:

- o módulo de elasticidade. OBS.: Se for digitado o valor zero nesta linha, adota-se os valores padrões que são:módulo de elasticidade = 210000 e módulo de Yong = 0.3
- o módulo de Yong. (caso não tenha digitado o valor zero na linha acima)

(apenas digite zero ou os valores do módulo de elasticidade e do módulo de Yong. Não digite o número do material)

Número de barras que compõem o modelo

Para cada barra forneça:

- o número da barra;
- o número do nó inicial;
- o número do nó final;
- o número do grupo de seções;
- o número do tipo de material;
- o ângulo de rotação entre a seção transversal da barra e o eixo local X.

Número de tipos de carregamento

Para cada carregamento digite:

- O número de nós com carga

Para cada nó com carga forneça:

- o número do nó;
- a força na direção do eixo X;
- a força na direção do eixo Y;
- a força na direção do eixo Z;
- o momento que provoca rotação em torno do eixo X;
- o momento que provoca rotação em torno do eixo Y;
- o momento que provoca rotação em torno do eixo Z.

- O número de barras com carga

Para cada barra com carga forneça:

- o número da barra;

- o tipo de carga → as opções são: D para carga uniformemente distribuida; T para carga linearmente distribuida e E para carga distribuida ao longo do eixo da barra.

- os dados da carga → para cada tipo deve-se

tipo D → fornecer o valor da carga e o ângulo que esta forma com o eixo local Y.

tipo T → fornecer o valor da carga e o ângulo que esta forma com o eixo local Y.

tipo E → fornecer o valor da carga.

Número combinações entre os tipos de carregamento

Para cada combinação digite:

- O número de tipos de carregamento que fazem parte da combinação

- para cada tipo de carregamento forneça sua posição na lista de tipos de carregamentos e o seu peso.

REFERENCIAS BIBLIOGRÁFICAS

AKÖZ, A. Y. & OMURTAG, M. H. and DOGRUOGLU, A. N. The mixed finite element formulation for three-dimensional bars. International Journal of Solids and Structures. vol 28 (2): 225-234, (Fev). 1991.

ASSAN, A. E. Método dos elementos finitos - primeiros passos. Universidade Estadual de Campinas, Departamento de Construção Civil - Faculdade de Engenharia Civil. (Mai).1993, p.59.

BLANDFORD, G. E. Stability analysis of flexibly connected thin-walled space frames. Computers & Structures. vol 53 (4): 839-847, 1994.

BREBBIA, C. A. & FERRANTE, A. J. The finite element technique. Porto Alegre. Editora da URGS - Universidade Federal do Rio Grande do Sul. 1975, p.410.

EMKIN, L. Z. Computers in structural engineering practice: the issue of quality. Computers & Structures. vol 30 (3): 439-446, (Mar). 1988.

MACKIE, R. I. Object oriented programming of the finite element method. International Journal for Numerical Methods in Engineering. vol 35 : 425-436, (Ago) .1992.

McGUIRE, W. Computers and steel design. Engineering Journal. vol 29 (4): 160-169, (Dez).1992.

NAJAFI, F. T. The computer in the construction industry. Computers & Structures. vol 41 (6): 1125-1132, (Dez). 1991.

PAPPAS, C. H. & MURRAY, W. H., Turbo C++ Completo e Total. São Paulo. Makron Books do Brasil Editora Ltda. 1991, p.771.

PETZOLD, C., Programando para Windows 3.1. São Paulo. Makron Books do Brasil Editora Ltda. 1993, p.1034.

SAADA, A. S. Elasticity Theory and Applications. New York. Pergamon Press Inc. 1974, p.351.

SCHOLZ, S. -P. Elements of an object-oriented FEM++ program in C++. Computers & Structures. vol 43 (3): 517-529, (Mai). 1992.

SOUSA, J. L. A. de O e. Determinação da carga crítica de flambagem em vigas contínuas sobre apoios elásticos contínuos e discretos. Seminário apresentado como trabalho final do curso de flambagem da Escola Politécnica da USP. São Paulo, 1986.

SWAN, T. Aprendendo C++. Rio de Janeiro. Editora Campus Ltda. 1993, p.675.

TOTTENHAM, H. & BREBBIA, C. Finite element techniques in structural mechanics. (?).[?], p.301.

VIZOTTO, I. Análise matricial de estruturas - conceitos fundamentais. Faculdade de Engenharia Civil da Unicamp, 1994. 37p. (Notas de Aula)

WATSON, A. S. and CHAN, S. H. A prolog-based object oriented engineering DBMS. Computers & Structures. vol 40 (1): 11-21, (Jun). 1991.

WEISKAMP, K. & HEINY, L. & FLAMING, B., Programação Orientada Para Objeto Com Turbo C++. São Paulo. Makron Books do Brasil Editora Ltda. 1993, p. 474.

GERE, J. M. & WEAVER, W., JR,. Análise de Estruturas Reticuladas. Rio de Janeiro. Editora Guanabara,1987. p. 443.

TIMOSHENKO, S. P. & GERE, J. E. Mecânica dos sólidos. Rio de Janeiro. LTC - Livros Técnicos e Científicos Editora S.A. 1984, Vol. II, p. 257-450.

MANDUJANO, M. L. S. Desenvolvimento de classes de matrizes para aplicação no método dos elementos finitos. Relatório encaminhado à FAPESP - processo número 93/0241-2. (Mai).1994, p.95.

BIBLIOGRAFIA CONSULTADA

PAPPAS, C. H. & MURRAY, W. H., Turbo C++ Completo e Total. São Paulo. Makron Books do Brasil Editora Ltda. 1991, p.771.

TOTTENHAM, H. & BREBBIA, C. Finite element techniques in structural mechanics. (?).[?], p.301.

BREBBIA, C. A. & FERRANTE, A. J. The finite element technique. Porto Alegre. Editora da URGS - Universidade Federal do Rio Grande do Sul. 1975, p.410.

SAADA, A. S. Elasticity Theory and Applications. New York. Pergamon Press Inc. 1974, p.351.

WEISKAMP, K. & HEINY, L. & FLAMING, B., Programação Orientada Para Objeto Com Turbo C++. São Paulo. Makron Books do Brasil Editora Ltda. 1993, p. 474.

DHATT, G. & TOUZOT, G., Une présentation de la méthode des éléments finis. Paris. Maloine S. A. Éditeur. 1981, p. 543.

O'BRIEN, S. K., Turbo Pascal 6 Completo e Total. São Paulo. Makron Books do Brasil Editora Ltda.1993, p.716.

FRANÇA, J. L. et all. Manual Para Normalização de Publicações Técnico-Científicas. Belo Horizonte. Editora UFMG, 1990. p. 168.

GERE, J. M. & WEAVER, W., JR,. Análise de Estruturas Reticuladas. Rio de Janeiro. Editora Guanabara,1987. p. 443.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. NBR 6023. Ago/1989. Referências Bibliográficas - Procedimento. p. 19.

ZIENKIEWICZ, O. C. Computational mechanics today. International Journal for Numerical Methods in Engineering. vol 34 (1): 9-33, (Mar) .1992.

PELEGRINO, S. and CALLADINE, C. R. Matrix analysis of statically and kinematically indeterminate frameworks. International Journal of Solids and Structures. vol 22 (4): 409-428, (Abr). 1986.

McGUIRE, W. Computers and steel design. Engineering Journal. vol 29 (4): 160-169, (Dez).1992.

ASSAN, A. E. Método dos elementos finitos - primeiros passos. Universidade Estadual de Campinas, Departamento de Construção Civil - Faculdade de Engenharia Civil. (Mai).1993, p.59.

NAJAFI, F. T. The computer in the construction industry. Computers & Structures. vol 41 (6): 1125-1132, (Dez). 1991.

WILSON, E. L. The use of minicomputers in structural analysis. Computers & Structures. vol 12 (5): 695-698, (Nov). 1980.

KANOK-NUKULCHAI, W. On a microcomputer integrated system for structural engineering practices. Computers & Structures. vol 23 (1): 33-37, (Jul). 1986.

EMKIN, L. Z. Computers in structural engineering practice: the issue of quality. Computers & Structures. vol 30 (3): 439-446, (Mar). 1988.

MEHRINGER, V. & PIERSON, G. and ORBISON, J. G. Computer-aided analysis and design of steel frames. Engineering Journal. vol 22 (3): 143-149, (Set).1985.

LI, C. Pretap: an expert preprocessor for a structural analysis program of tall buildings. Computers & Structures. vol 33 (3): 897-902, (Mar). 1989.

PETZOLD, C., Programando para Windows 3.1. São Paulo. Makron Books do Brasil Editora Ltda. 1993, p.1034.

WATSON, A. S. and CHAN, S. H. A prolog-based object oriented engineering DBMS. Computers & Structures. vol 40 (1): 11-21, (Jun). 1991.

CLOUGH, R. W. The finite element method after twenty-five years: a personal view. Computers & Structures. vol 12 (4): 361-370, (Out). 1980.

THEOCARIS, P. S. and KARAYANOPOULOS, N. An algorithm for the numerical solution of dense large general linear systems. Computers & Structures. vol 14 (5-6): 377-383, (Dez). 1981.

PESQUERA, C. I. & McGUIRE, W. and ABEL, J. F. Interactive graphical preprocessing of three-dimensional framed structures. Computers & Structures. vol 17 (1): 1-12, (Jul). 1983.

SCHOLZ, S. -P. Elements of an object-oriented FEM++ program in C++. Computers & Structures. vol 43 (3): 517-529, (Mai). 1992.

MACKIE, R. I. Object oriented programming of the finite element method. International Journal for Numerical Methods in Engineering. vol 35 : 425-436, (Ago) .1992.

SCHOLZ, H. and FALLER, G. A micro-computer program for the elastic-plastic analysis and optimum design of plane frames. Computers & Structures. vol 24 (6): 941-947, (Jun). 1986.

LEUNG, A. Y. T. Micro-computer analysis of three-dimensional tall buildings. Computers & Structures. vol 21 (4): 639-661, (Out). 1985.

RESENDE, L. and DOYLE, W. S. "Nonpri" - An effective non-prismatic three dimensional beam finite element. Computers & Structures. vol 14 (1-2): 71-77, (Out). 1981.

BEDROSIAN, G. Shape functions and integration formulas for three-dimensional finite element analysis. International Journal for Numerical Methods in Engineering. vol 35 : 95-108, (Jul) .1992.

AKÖZ, A. Y. & OMURTAG, M. H. and DOGRUOGLU, A. N. The mixed finite element formulation for three-dimensional bars. International Journal of Solids and Structures. vol 28 (2): 225-234, (Fev). 1991.

GATTASS, M. & PAULINO, G. H. and GORTAIRE C., J.C. Geometrical and topological consistency in interactive graphical preprocessors of three-dimensional framed structures. Computers & Structures. vol 46 (1): 99-124, (Jan). 1993.

SUBRAMANIAN, N. and CHETTIAR, C. G. The computer analysis of space frames with offset members. Computers & Structures. vol 11 (4): 297-303, (Abr). 1980.

PAULINO, Gláucio Hermógenes. Preprocessamento de estruturas reticuladas espaciais, com reordenação nodal, usando iteração gráfica interativa. Rio de Janeiro, 1988. 298p. / Dissertação - Mestrado - Departamento de Engª Civil da Pontifícia Universidade Católica do Rio de Janeiro./

SOUSA, J. L. A. de O e. Determinação da carga crítica de flambagem em vigas contínuas sobre apoios elásticos contínuos e discretos. Seminário apresentado como trabalho final do curso de flambagem da Escola Politécnica da USP. São Paulo, 1986.

VIZOTTO, I. Análise matricial de estruturas - conceitos fundamentais. Faculdade de Engenharia Civil da Unicamp, 1994. 37p. (Notas de Aula).

SPILLERS, W. R. Geometric stiffness matrix for space frames. Computers & Structures. vol 36 (1): 29-37, 1990.

BLANDFORD, G. E. Stability analysis of flexibly connected thin-walled space frames. Computers & Structures. vol 53 (4): 839-847, 1994.

SANAL, Z. Finite element programming and C. Computers & Structures. vol 51 (6): 671-686, 1994.

ZEGLINSKI, G. W. and HAN, R. P. S. Object oriented matrix classes for use in a finite element code using C++. International Journal for Numerical Methods in Engineering. vol 37 (22): 3921-3937, (Nov) .1994.

ARGYRIS, J. H. et all. On the geometrical stiffness of a beam in space - a consistent V.W. approach. Computer Methods in Applied Mechanics and Engineering. vol 20 (1): 105-131, (Oct) .1979.

DUBOIS-PÈLERIN, Y. and ZIMMERMANN, T. Object-oriented finite element programming: III. An efficient implementation in C++. Computer Methods in Applied Mechanics and Engineering. vol 108 (1-2): 165-183, (Sep) .1993.

SWAN, T. Aprendendo C++. Rio de Janeiro. Editora Campus Ltda. 1993, p.675.

TIMOSHENKO, S. P. & GERE, J. E. Mecânica dos sólidos. Rio de Janeiro. LTC - Livros Técnicos e Científicos Editora S.A. 1984, Vol. II, p. 257-450.

WILSON, E. L. & HABIBULLAH, A. SAP-80 Structural Analysis Programs. A series of computer programs for the static and dynamic finite element analysis of structures Berkeley. Computers & Structures Inc. 1984, p. 193.

MANDUJANO, M. L. S. Desenvolvimento de classes de matrizes para aplicação no método dos elementos finitos. Relatório encaminhado à FAPESP - processo número 93/0241-2. (Mai).1994, p.95.